Two Search Techniques for Imperfect Information Games and Application to Hearts

Theodore J. Perkins

CMPSCI Technical Report 98-71

May 1998

NOTE: This paper is available by anonymous ftp from the site **ftp.cs.umass.edu** in the directory **pub/techrept/techreport/1998**

Two Search Techniques for Imperfect Information Games and Application to Hearts

Ted Perkins
perkins@cs.umass.edu

Abstract

Two techniques for game-tree search in imperfect information games with an arbitrary number of players are described: II-Max_n and MC-Max_n. They use probability computations and Monte Carlo-style sampling to overcome imperfect information problems. We then discuss implementations of both algorithms for the card game Hearts. We describe and empirically evaluate methods for estimating probabilities relevant to card games, and report on the performance of the search procedures. We also compare the Hearts-playing ability of the programs against each other and against a rule based Hearts player.

Introduction

Game tree search has been a very successful paradigm for computer playing of perfect-information games. To name just two, Deep Blue (Hsu, Campbell, & Hoane 1995) and Logistello (Buro 1997) have been enormously successful in chess and Othello respectively – beating the best human players. Games with a stochastic element, such as a die roll, can also be handled successfully by algorithms such as Expectimax (Russell & Norvig 1995) as long as a distribution for the random element is known. Alternatively, one could develop an extremely good heuristic function for evaluating moves with little or no search, overcoming problems of randomness and large branching factor simultaneously. This method was used by Tesauro in creating his world-class Backgammon player TD-Gammon (Tesauro 1995).

In some games, the players have only imperfect information – they do not have complete knowledge of their current situation. For instance, in most card games a player will not know which cards are held by the other players. Imperfect information is problematic for the standard game tree search methods.

In the usual formulation of game tree search, at each internal node we recursively evaluate each move the player might make. In an imperfect information game, we may not know which moves are available to the other players, or moves may have different probabilities of availability. Further, heuristic evaluations of the leaves of the tree must produce their predictions in the face of only partial information.

We describe two different algorithms, II-Max_n and MC-Max_n, for search in imperfect information games. The idea of MC-Max_n has been circulating unnamed in the game-playing literature for the past decade. Wheen (1989) and Levy (1989) suggested its use for Bridge, and Ginsberg (1996a; 1996b) has an implementation. Although neither of II-Max_n and MC-Max_n are designed for games with a stochastic element, they can easily be generalized to do so in the same way that Expectimax generalizes Minimax. We discuss implementations of these two algorithms for the card game Hearts, and present empirical studies of the algorithms' performance.

Search Algorithms

For generality and because of our target application, we have designed the search methods to allow games with an arbitrary number of players. The outcome of such a game is given by a payoff vector, with each element representing the final outcome of the game for a different player. The payoff vector formulation also means that we are not limited to zero- or constant-sum games. Our only assumption is that each player seeks to maximize his own element in the final payoff vector, without regard to the payoffs of the other players.

II-Max_n

The Max_n algorithm (Luckhardt & Irani 1986) extends Minimax to the multiple-player case. Leaves are evaluated to payoff vectors, and the vectors are propagated up the tree using the assumption that each player maximize his own payoff. Like Max_n, II-Max_n builds a single game tree to evaluate the possible moves of the root player. However, the generation of moves at internal nodes, and the recombination of the payoff vectors for each move is different.

At an internal node of the game tree, we recursively *evaluate* each move the player might make. Since we do not know the true state of the game, II-Max_n generates a list of *all moves* available to the player in *any* game state consistent with the information we have. Many of these moves may be mutually exclusive; in fact, in our Hearts implementation it is not uncommon to search more moves than the player has cards. In order for us to correctly compute the expected payoffs at an internal node, every move the player might possibly make must be searched.

After these moves are recursively evaluated, we must combine their payoff vectors to form an evaluation of the internal node. If the highest-valued move is available, then we should expect its payoff vector. If that move is unavailable, but the second highest-valued move is available then its payoffs are what we should expect, and so on.

Suppose that the player has N moves; let $V_1, V_2, \ldots, V_N$ be the payoff vectors for these moves, sorted by decreasing value for the current player, and let P_i for $i \in 1 \ldots N$ be the probability that the i^{th} highest-valued move is available but no higher valued moves are; then the value of the current node is:

$$V = V_1 * P_1 + V_2 * P_2 + \ldots + V_N * P_N$$

For the root player we presumably know exactly which moves are available, so $P_1 = 1$ and $P_i = 0$ for $i > 1$, and the equation degenerates to taking the move with maximum value.

II-Max_n requires that we search *any* move consistent with our imperfect knowledge of the game state. For some games this set of moves may be impractically large. In fact, this is a difficulty in our application of II-Max_n to Hearts; despite theoretical incorrectness, we use forward-pruning heuristics to reduce the branching factor to a manageable level. For II-Max_n to be applicable, one must also be able to compute or estimate the P_is. II-Max_n is most appropriate when these probabilities are not too difficult to compute, and when the number of potential moves (which determines branching factor) is reasonable.

MC-Max_n

An alternative way to get at the value of a root-level move is as its Max_n value, averaged over all game states consistent with our imperfect knowledge. This suggests a Monte Carlo approach to move evaluation. For as many times as we care:

1. Generate a complete (i.e. perfect information) game state consistent with our imperfect knowledge.
2. Perform a Max_n search to evaluate each available move.

   And when we have sampled and searched enough times:

3. Choose the move that has the highest average payoff.

We want to sample enough times to get a reliable estimate of the values of possible moves; this number can depend on the game we are playing and our current situation in the game. We may also be concerned about meeting time constraints.

Because II-Max_n requires us to search every possible move no matter how unlikely, and probabilities for each move must be computed, an II-Max_n tree will be large and slow to compute. MC-Max_n builds simpler, perfect information trees, but it may have to do so many times to get good estimates of the values of different moves. Depending on characteristics of the game being played, either method might be better. We do note that MC-Max_n offers extra flexibility for time-sensitive applications because the number of samples taken can be varied in addition to the search depth.

Bluto and Sweetpea

We tested both of these methods by applying them to the card game Hearts. Hearts is a trick-based game, played like no trumps Bridge. The cards are dealt to the players, and the player holding the two of clubs begins play by leading that card. Players must follow suit if possible. The player that plays the highest card in the lead suit wins the trick and leads the next trick. The only restriction on subsequent leads is that Heart cards and the Queen of Spades may not be lead until one of them has been played (as a follow) in a previous trick.

Unlike Bridge, scoring is based on cards taken during tricks, not the number of tricks taken. Each Heart card taken subtracts one point from the trick winner's score, and taking the Queen of Spades subtracts 13 points. The exception is that if a player takes *all* of the Hearts and the Queen in a single hand, then each of the *other* players loses 26 points. This feat is called "Shooting the moon."[1] We also play with the Jack of Diamonds rule – the player who takes this card has 10 points added to his score. Each hand of Hearts is thus a multi-player game with imperfect information, no random elements after the initial deal, and is not constant-sum due to the possibility of shooting the moon.

A *game* of hearts is a set of hands, played until one of the players reaches some predetermined score.

[1] Alternatively, some people prefer to add 26 points to a player who shoots the moon, and leave the other players' scores unchanged.

We use -75 points, the default set by Paul Utgoff for four-player games on his Internet Hearts ladder. (Note *that the total* points distributed in each hand is -16 or below, so scores tend downwards.) Bluto and Sweetpea do not attend to any information about the *game* scores, they just try to maximize the points they get out of each hand individually.

Hearts may be played with three to five players, but we focus on four player games in this paper. Readers familiar with Hearts will know that there is also a passing phase to the game, after the initial deal but before the playing of tricks begins. We do not dwell on passing here, as it is unrelated to the search procedures that are the topic of this paper.

Bluto's Search

Bluto uses II-Max_n to play hearts. Each node in the tree consists of what Bluto knows: the cards it holds and what it has deduced about the other players' hands, which cards have been taken by which players in prior tricks, and possibly information about a trick in progress. To organize the knowledge about the players' hands, Bluto maintains what we call a YMN table. For each player and for each card in the deck it stores a "Yes", "Maybe", or "No," meaning that the player definitely does, may, or definitely does not hold the card. Branches of the game tree correspond to different cards the players might play.

Bluto uses two different degrees of forward pruning to limit the branching factor; this was found necessary if search is to complete in reasonable time. When a player is leading a new trick, all possible leads are considered. When a player is following suit, the less-pruned method is to search in-suit all cards the player might (or does) have. An effective pruning heuristic for in-suit follows is to consider up to four cards: the lowest and highest cards that might be held in the lead suit, and the cards closest above and below the current card that is winning the trick. Naturally, if some of these cards coincide they are not searched twice. If the chance that the player is void in the lead suit exceeds a threshold, we consider up to five other follows: the Queen of Spades and Jack of Diamonds, if there is some chance they are held, and the highest card in each non-lead suit.

Bluto has an unusual method for computing the probabilities that a player holds a certain cards – probabilities required for the P_is used in recombining the payoff vectors of possible plays. Exact computation is too difficult a combinatorial problem, so we use an iterative approximation scheme; since the probabilities of holding various cards are interdependent, we actually compute a whole table of probabilities with an entry for each player and card. The algorithm follows:

1. Initialize the probability table from the YMN table. A "yes" gets probability 1, and "no" gets 0, and a "maybe" gets $(H_p - Y_p)/M_p$, where H_p is the total number of cards held by player p, Y_p the number of cards we know he has, and M_p the number of cards he might have. In words, the count of unidentified cards held by the player is spread evenly over all the "maybe"s.

   Improve our Approximation:

2. For each player p, let S_p be the sum of the all the card probabilities for p. This should sum to the number cards held by the player, H_p. If not, add to each probability corresponding to a "maybe" the quantity $\alpha * (H_p - S_p)$.

3. For each card c, let S_c be the sum of the probabilities for that card across all players. This should sum to 1. If not, add to each "maybe" $\alpha * (1 - S_c)$.

4. If we choose, further refine our approximation by going to step 2. Otherwise, we are done.

Here, α is an update rate – we found that 0.5 produced the fastest convergence while maintaining stability. We may terminate the process either by noting when the probabilities are not changing very much, or by simply completing a specified number of iterations. We found that after 10 iterations there is no significant improvement in accuracy, but fewer iterations are insufficient. In the experimental section we report on this method's accuracy and speed.

Sweetpea's Search

Sweetpea uses MC-Max_n search to evaluate possible plays. The game tree node contents are the exact hands of each player, outcomes of prior tricks, and information on the trick in progress, if any. Care must be taken in generating the perfect information samples of the game state to which Max_n is applied. It is easy to come up with a method that does not sample uniformly from the set of states consistent with our knowledge. Sweetpea starts from the same YMN table that Bluto uses, and performs the following algorithm:

1. If no "maybe"s remain, we have generated a perfect information state and are done. Otherwise, select a "maybe" from the table uniformly randomly. Let this "maybe" be for player p and card c.

2. Set entry (p, c) to "yes" and draw any other conclusions that may be made. For instance, (q, c) for players $q \neq p$ are all "no". It may also be that some player now has as many "yes"s as cards held,

so all "maybe"s for that player should be set to "no". On the other hand, a player may have as *many* "maybe"s and "yes"s as cards held, thus all "maybe"s should be set to "yes". Goto step 1.

Although we do not prove mathematically that this is an unbiased method of sampling the consistent states, we empirically found it to be so. We were able to compute exact card probabilities for certain YMN tables where not too much was unknown – mainly in tricks near the ends of hands[2]. We then used the sampling method to estimate the card probabilities by generating many (up to 100,000) consistent states and counting how many times a card appeared in a particular player's hand. These probabilities were seen to converge to the exact probabilities, so we claim the sampling method is correct.

Evaluation Functions

Both Bluto and Sweetpea have heuristic evaluation functions that apply only between tricks. This restricts the search procedure because we cannot terminate at any ply we want; we can only end search at a trick-ending ply. This decision simplifies the evaluation function, however, in that no information about an ongoing trick need be taken into account.

Sweetpea evaluates perfect information situations at the leaves of its search tree. The details of our evaluation function change as we improve it, but the first step is always to compute a number of simple counting-style features. We tabulate the numbers of high and low cards held, and in particular the numbers of cards held by each player above and below the Queen of Spades and the Jack of Diamonds. We count void suits, and the number of tricks in a row that each player can win in each suit assuming the player has the lead. We also have a "softer" version of this last feature; assuming that a player gets to lead and that each opponent will choose in-suit follows randomly, we compute the expected number of tricks that can be won in each suit. This feature bears on how many tricks a player can win, which affects his ability to shoot the moon. Using these features, we then estimate the probability of a successful shoot and the expected outcome of choosing not to shoot. These estimates are conditioned on who holds certain key cards, and based partly on programmer intuition and tuned by observing of Sweetpea's play.

Luckhardt and Irani discussed pruning for the Max_n algorithm based on evaluating only some components of the payoffs vectors at leaves (Luckhardt & Irani 1986). Sweetpea's heuristic evaluator is such that single components of the payoff vector may be computed, so pruning is applicable here. This would required significant bookkeeping and rearranging of the search procedure, and has not yet been implemented. Informal testing suggested that Sweetpea does spend a significant amount of time evaluating its leaves, so we might expect to gain a lot from such pruning.

Bluto was originally designed with its own evaluation function, but in order for the comparisons in the experiments section to be more meaningful we are currently running Bluto with Sweetpea's evaluation function. At a leaf, Bluto creates a sample set of perfect information states in the same way Sweetpea creates its samples. It applies Sweetpea's evaluator to each and averages the results to form an expected payoff vector.

Experimental Results

In this section we compare different methods for estimating card probabilities, report on search tree characteristics and evaluate Bluto's and Sweetpea's performance.

Probability Estimation

The computation of card location probabilities is an explicit part of Bluto, and implicit in Sweetpea's sampling and subsequent search. In this section we compare the accuracy and speed of several methods for estimating these probabilities. The results reported are for a standard test set of 455 game situations. These were generated by playing 4 Sweetpeas against each other for three games, a total of 35 hands. At the beginning of each trick, the Sweetpea in the first position recorded its world view – the cards it held, and all it had deduced about the other players' hands and seen in previous tricks. These situations are intended to represent realistic, naturally occurring situations for which probabilities need to be calculated. The methods we report on are:

1. Exact computation of probabilities, which is feasible only in the later tricks in a game, when little is unknown.

2. Iter10: Bluto's Iterative algorithm, run for 10 iterations. This number was chosen to minimize computation time, while still achieving near to the algorithm's asymptotic accuracy.

3. Samp10, Samp150: We sample the imperfect information state 10 (resp. 150) times, and see how often each card is held by each player. In order to meet

[2] A simple procedure that tentatively assigns cards to a certain player and recursively counts how many states are consistent with this assignment is all that is needed to compute exact probabilities.

Figure 1: Error in Probability Estimates

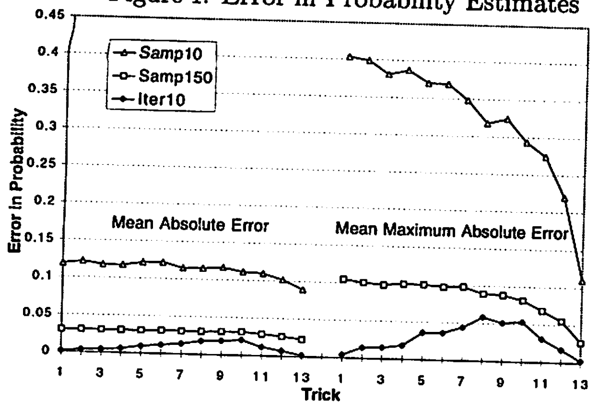
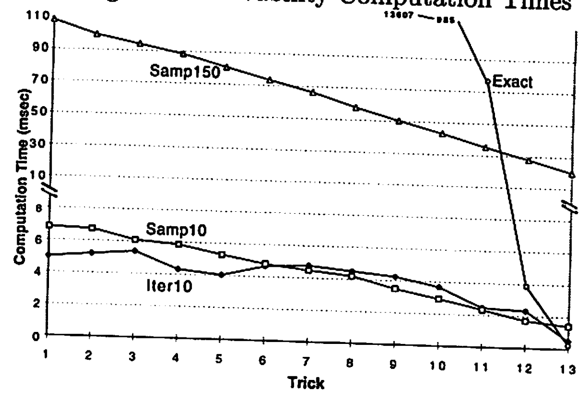


Figure 2: Probability Computation Times



timing constraints, Bluto uses 10 samples to evaluate leaves and Sweetpea searches 150 samples, hence the sample sizes we report on here.

Figure 1 displays the error in the probabilities estimated by these methods compared to the estimates based on 100,000 samples, which we believe are quite close to correct. On the left side of the graph we see the mean absolute error in the probability table, averaged over uncertain cards and all hands, but separated by trick. The mean maximum absolute error, on the right side, is the largest error in absolute value for any player and card in the table, averaged across hands.

Samp10's errors are quite large, but perhaps not unexpectedly so. The probabilities are off by about .1 on average. More alarming is the mean maximum error, a whopping .3-.4 during most of the game. Samp150's estimates are much better, but not as good as we would like. Apparently Samp's convergence to the correct values is rather slow. The iterative method, surprisingly, has the least error of all, uniformly beating the other two methods.

Iter10 is also significantly faster than the other methods. Figure 1 shows mean computation times, by trick, for the four different methods. Iter10 and Samp10 are equally speedy, with Samp150 a distant third (note the split scale on the time axis). Iter10 is more accurate than Samp150 and as fast as Samp10, so we use it in Bluto despite its asymptotic incorrectness. The inaccuracy of the estimates based on sampling raises concerns for both Sweetpea and Bluto. Are Bluto's leaf evaluations too unreliable because it is only sampling and evaluating the state 10 times? Is Sweetpea testing enough Max_n trees to get good estimates? We return to these questions later.

Search Characteristics

On the same sample of 455 game situations we tested three search procedures: Bluto with the lesser and greater levels of forward pruning and 10 samples to evaluation every leaf and Sweetpea at 150 samples. We refer to these as Bluto-lp (less pruning), Bluto-mp (more pruning) and Sweetpea-150. We searched each position as if we were leading the trick – searching to the end of the trick for a total of four ply[3]. We report the results as a function of trick number, giving us some idea of total cards held and decreasing uncertainty about the game state affect search.

Figure 3 illustrates the mean search time per trick. We desired an average time of 10 seconds per move, and Bluto-mp's and Sweetpea-150's sample sizes were set to meet this criterion. Bluto-lp takes considerably longer at 10 samples per leaf, with time increasing superlinearly as trick number decreases.

We graph the mean branching factors for the same set of searches in Figure 4. Bluto-mp and Sweetpea-150 have fairly similar branching factors, and show a flat profile in the early tricks tapering to zero at the end of the game[4]. Bluto-lp has a branching factor roughly twice that of the others and it increases significantly toward the beginning of the game, so Bluto-lp's long search time is not surprising.

Players' Performance

Figure 5 compares three different hearts-playing programs set against each other in various arrangements. The letters s, b and f stand for the players Sweetpea, Bluto-mp and f86 respectively. F86 is a rule-based hearts-playing program written by Paul Utgoff in 1986; it was the winner of his computer Hearts tournaments

[3]Four ply may not seem like much by contemporary standards, but realize that Sweetpea, for instance, is doing a four ply search 150 times. The total work is comparable to a single 7-8 ply Max_n search

[4]All of the search procedures begin by counting the number of moves available to the searching player. If there is only one, then no searching is done; hence the branching factors go to zero in the final trick.

Figure 3: Mean Search Times

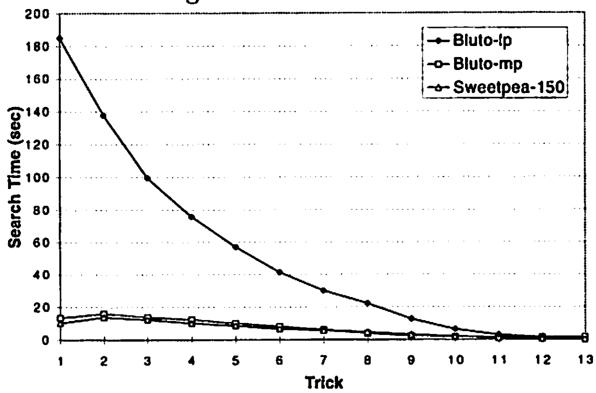
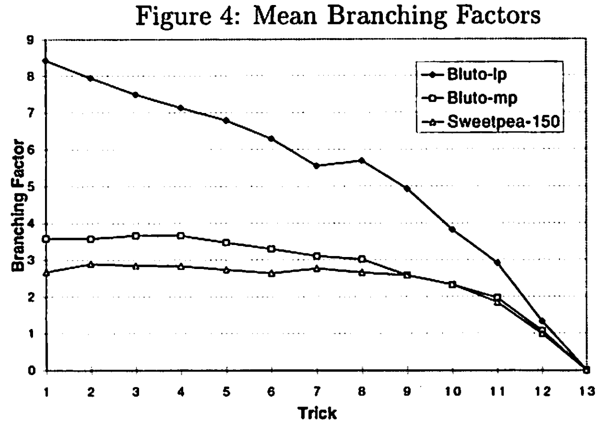


Figure 4: Mean Branching Factors



then and until recently. All games had four players, and ended when any player dropped below -75 points. We report each player's mean score and mean rank (1 for highest score in game, down to 4 for lowest score) averaged over 100 games.

A 95% confidence interval for the game scores has a radius of 5-6 points, and .2 for the ranks. So, for instance, in the first row of either table we see that Sweetpea significantly outplays all of the three f86s set against it. In only one case, the first and last players in the fourth pattern, does an f86 achieve a better score than a Sweetpea, and the difference is negligible. Nearly all of the possible pairwise Sweetpea/f86 comparisons show Sweetpea playing significantly better. Sweetpea does best when it is in a game alone; as the number of Sweetpeas increases, their scores become depressed. We attribute this to Sweetpea's conservative, defensive style of play. When it is alone, it can defend itself, but when everybody is playing defensively then no one does very well.

Bluto, using the same evaluation function but a different search procedure, plays worse than f86. In about two thirds of the possible comparisons, Bluto does significantly worse in terms of score. Looking at the mean ranks puts Bluto in a slightly better light. Even after 100 games, the Bluto in the b,f,f,f pattern is not losing statistically significantly more games than any of the f86s. The Sweetpea/Bluto games' outcomes are similar to the Sweetpea/f86 games.

There are several possible explanations for Bluto's inferior play: (a) too much forward pruning, (b) inaccurate probability computation, (c) too few samples for the leaf evaluations, (d) Sweetpea's evaluation function does not work well with Bluto's search procedure. We believe that we have not pruned important plays, and we intend to test (b) by substituting an unbiased monte carlo probability estimation into Bluto's code. Such a version could not be used in competition because it would be too slow, but would help to clarify the source of Bluto's weakness.

We did examine the effect of sample size on both Sweetpea's and Bluto's performance, with the results displayed in Figure 6. The lines represent the mean game score on 100 games of Sweetpea against three f86s and Bluto against the same. 95% confidence intervals are plotted along with each data point.

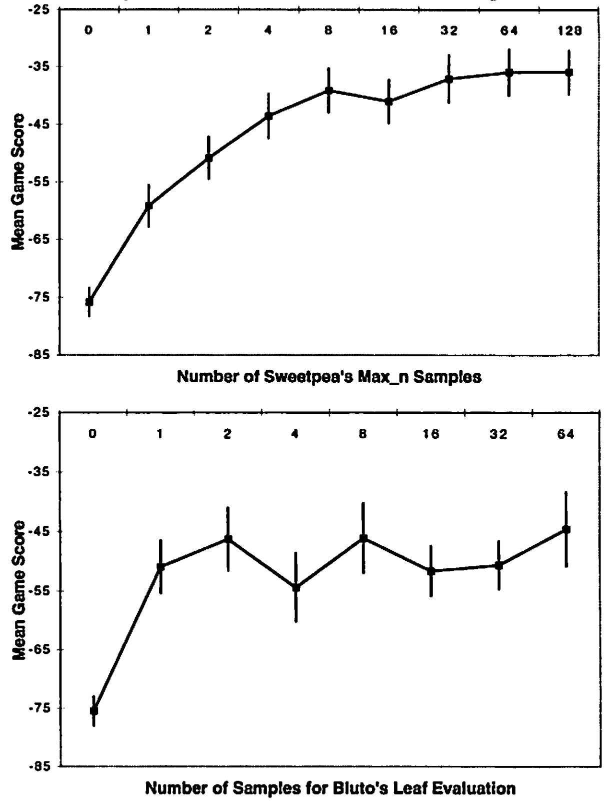
Sweetpea's performance increases nicely along with the number of samples, seeming to level off as we reach 64 samples. The 150 samples used in the table of matches above should be more than enough. Bluto seems strangely insensitive to the number of samples it makes at its leaves; it could be that Sweetpea's evaluation function, which was tuned to get good perfor-

Figure 5: Play Comparison

players	score			
p1 p2 p3 p4	p1	p2	p3	p4
s,f,f,f	**-35.2**	-52.3	-50.1	-54.8
s,s,f,f	-45.3	**-42.0**	-50.7	-56.4
s,f,s,f	**-41.1**	-50.2	**-41.4**	-53.9
s,s,s,f	-52.7	-48.7	**-40.6**	-52.4
b,f,f,f	-53.35	-46.32	-51.12	**-46.00**
b,b,f,f	-55.23	-48.41	**-42.66**	-51.46
b,f,b,f	-52.38	**-40.7**	-54.7	-40.74
b,b,b,f	-48.84	-43.4	-52.26	**-43.02**
s,b,b,b	**-32.6**	-49.9	-52.3	-48.2
s,s,b,b	-36.8	**-36.4**	-53.8	-56.9
s,b,s,b	-39.2	-54.9	**-36.4**	-54.0
s,s,s,b	**-42.8**	-43.7	-44.4	-52.4

players	rank			
p1 p2 p3 p4	p1	p2	p3	p4
s,f,f,f	**2.02**	2.63	2.55	2.76
s,s,f,f	2.38	**2.19**	2.58	2.82
s,f,s,f	2.34	2.55	**2.29**	2.79
s,s,s,f	2.60	2.59	**2.17**	2.59
b,f,f,f	2.54	**2.38**	2.56	2.44
b,b,f,f	2.71	2.36	**2.24**	2.63
b,f,b,f	2.69	2.30	2.73	**2.25**
b,b,b,f	2.68	2.36	2.68	**2.27**
s,b,b,b	**1.98**	2.65	2.72	2.62
s,s,b,b	**2.09**	2.28	2.69	2.92
s,b,s,b	2.24	2.87	**2.11**	2.73
s,s,s,b	2.38	**2.32**	2.43	2.79

Figure 6: The Importance of Sample Size



mance out of Sweetpea, is ill-suited to Bluto's search procedure.

In an informal attempt to get a broader-based evaluation of our programs, we tested Sweetpea against a publicly available shareware program for the Macintosh, Hearts Deluxe. The scores on three games were: -29,-71,-71,-101; -29,-71,-100,-24; and -32,-70,-95,-103 where the first score in each set is Sweetpea's. A moderately-skilled human player is generally able to beat any of these computer players, especially after some practice games to characterize their strategy and find weaknesses. Computer Hearts players have a ways to go before they achieve the dominance of Deep Blue or Logistello.

Conclusion

We have described two methods for search in games where the players have only imperfect information about the state of the game, II-Max_n and MC-Max_n. II-Max_n tries to build a game tree that accounts for incomplete knowledge probabilistically, and is close in conception to the Expectimax algorithm (Russell & Norvig 1995). The assumption that it is the availability of moves themselves that is uncertain requires a different method for combining the values of children in the search tree. Unfortunately, the required probability computations and branching factor make these trees ponderous. MC-Max_n builds a set of simpler, perfect information trees and averages the results; no probability computations are involved, and branching factor may be reduced as well.

Sweetpea, using MC-Max_n, played better Hearts than the II-Max_n player Bluto and the rule-based system f86. The lesser performance of Bluto is not entirely explained, but it may be due in part to inaccurate probability estimates in the tree. It may also be that the use of Sweetpea's evaluation function at Bluto's leaves, which was done mainly for comparison purposes, is inappropriate. We continue to work on improving the evaluation functions. We would like to use reinforcement learning to tune functions for Bluto and Sweetpea in a more principled way, and we want to add features to encourage more aggressive play, especially shooting the moon.

Card games offer and excellent venue for exploring issues in imperfect information search. Many control problems exhibit imperfect information, either because important variables are unobservable or are too costly to observe frequently (Bertsekas 1995). Game theorists and economists also study many competitive situations in which imperfect information plays a role (Shubik 1983). Search procedures for these types of problems have many potential applications.

References

Bertsekas, D. P. 1995. *Dynamic Programming and Optimal Control.* Belmont, Massachusetts: Athena Scientific.

Buro, M. 1997. The othello match of the year: Takeshi murakami vs. logistello. *ICCA Journal* 20(3):189–193.

Ginsberg, M. 1996a. Analysis of the law of total tricks. *The Bridge World.* Also available via ftp at ftp.cirl.uoregon.edu as /pub/users/ginsberg/papers/total.ps.gz.

Ginsberg, M. 1996b. How computers will play bridge. *The Bridge World.* Also available via ftp at ftp.cirl.uoregon.edu as /pub/users/ginsberg/papers/bridge.ps.gz.

Hsu, F.-H.; Campbell, M. S.; and Hoane, A. J. 1995. Deep Blue system overview. In ACM., ed., *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995,* Conference Proceedings of the International Conference on Supercomputing 1995; 9th, 240–244. New York, NY 10036, USA: ACM Press.

Levy, D. 1989. The million pound bridge program. In Levy, D., and Beal, D., eds., *Heuristic Programming in Artifical Intelligence: the first computer olympiad.* Ellis Horwood. 95–103.

Luckhardt, C. A., and Irani, K. B. 1986. An algorithmic solution of n-person games. In *Proceedings of the Fifth National Conference on Artificial Intelligence,* 158–162. Morgan Kaufmann.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach.* Upper Saddle River, New Jersey: Prentice Hall.

Shubik, M. 1983. *Game Theory in the Social Sciences: Concepts and Solutions.* Cambridge, Massachusetts: MIT Press.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Communications of the ACM* 38(3):58 – 68.

Wheen, R. 1989. Solving double dummy bridge problems by exhaustive search. In Levy, D., and Beal, D., eds., *Heuristic Programming in Artifical Intelligence: the first computer olympiad.* Ellis Horwood. 88–94.