Learning To Play Hearts

Nathan R. Sturtevant and Adam M. White

Department of Computer Science
University of Alberta
Edmonton, AB, Canada T6G 2E8
{nathanst, awhite}@cs.ualberta.ca

Abstract. Temporal difference (TD) learning has been used for learning in a variety of different games. TD-gammon illustrated how the combination of game tree search and learning methods can achieve grand-master level play in backgammon. However, this expert level of intelligence has been difficult to replicate in other games. In this work, we develop a hearts player based on a linear perceptron and TD learning. By carefully engineering a small set of basic game features, our simple learner was able to beat the best known search based hearts agents with little parameter tuning. We report initial results on learning with various combinations of features and training under self-play and against expert level search players.

1 Introduction and Background

Learning algorithms offer the promise of successfully solving complex control tasks. But, in practice there is a significant amount of work that needs to be done to tune each learning approach for specific problems and domains. We describe here the methods used to build a program to play the game of Hearts which is significantly stronger than a previously built expert-based program.

1.1 Hearts

Hearts is a trick-based card game, usually played with four players and a standard deck of 52 cards. Before play begins, 13 cards are dealt out to each player face-down. After all players look at their cards, the first player plays (leads) a card face-up on the table. The other players follow in order, if possible playing the same suit as was lead. When all players have played, the player who played the highest card in the suit that was led wins or *takes* the trick. This player places the played cards face-down in his pile of taken tricks, and leads the next trick. This continues until all cards have been played.

The goal of Hearts is to take as few points as possible. A player takes points based on the cards in the tricks taken. Each card in the suit of hearts (♡) is worth one point, and the queen of spades (Q♠) is worth 13. If a player takes all 26 points, also called *shooting the moon*, they instead get 0 points, and the other players get 26 points each.

For those familiar with the game of Hearts, there are many variations on the rules of Hearts for passing cards between players before the game starts, determining who leads first, determining when you can lead hearts, and providing extra points based on play. In this work we use the simplest variations of the rules; there is no passing of cards and no limitations on when cards can be played. Also, we do not use the rule for shooting the moon.

Hearts is an imperfect information game because players cannot see their opponents' cards. In practice, imperfect information in card games has been handled successfully using Monte-Carlo search ([1], [2]) despite the known limitations of this approach [3]. We concentrate on learning to play the perfect-information version of the game for use with Monte-Carlo sampling in the full game.

1.2 Hearts-Related Research

There have been several studies on learning and search algorithms for the game of Hearts. Because Hearts is a multi-player game (more than two players) minimax search cannot be used. Instead $\max^n$ [4] is the general algorithm for playing multi-player games.

Perkins [5] developed two search-based approaches for multiplayer imperfect information games based on $\max^n$ search. The first search method built a $\max^n$ tree based on move availability and value for each player. The second method maximized the $\max^n$ value of search trees generated by Monte-Carlo. The resultant players yielded low to moderate levels of play against human and rule-based based players.

One of the strongest computer Hearts programs [2] resulted from work focused primarily on algorithms for playing multi-player games. This program was significantly stronger than one of the best commercial programs available at the time. However, the program is still weaker than humans; its biggest weakness is its inability to stop opponents from 'shooting the moon'. We used this program as an expert to train against in some of our experiments for this paper.

Fujita *et al* have done several studies on applying reinforcement learning (RL) algorithms to the game of hearts. In their recent work [6], Fujita *et al* approached the game as a POMDP, where the agent can observe the game state, but other dimensions of the state, such as the opponents' hands, are unobservable. Using a one step history, they constructed a new model of the opponents' action selection heuristic according to the previous action selection models. Although their learned player performed better than their previous players and also beat rule-based players, it is difficult to know the true strength of the learning algorithm and resultant player. This is due to the limited number of training games and lack of validation of the final learned player. The work of Fujita *et al* differs from ours in that they understate the feature selection problem, which was a crucial factor in the high level of play exhibited by our learning agents.

Finally, Fürnkranz *et al* [7] have done some initial work using RL techniques to employ an operational advice system to play hearts. A neural network was used with temporal difference learning to learn a mapping from state abstraction

and action selection advice to a number of move selection heuristics. This allowed their system to learn a policy using a very small set of features (15) and little training. However, this work is still in its preliminary stages and the resulting player exhibited only minor improvement over the greedy action selection of the operational advice system on which the learning system was built.

1.3 Reinforcement Learning

In reinforcement learning an agent interacts with an environment by selecting actions in various states to maximize the sum of scalar rewards given to it by the environment. In hearts, for example, a state consists of the cards held by and taken by each player in the game and negative reward is assigned each time you take a trick with points on it. The environment includes other players as well as the rules of the game.

The RL approach has strong links to the cognitive processes that occur in the brain and has proved very effective in Robocup soccer [8], industrial elevator control and backgammon [9]. These learning algorithms were able to obtain near optimal policies simply from trial and error interaction with the environment in high dimensional and sometimes continuous valued state and action spaces.

All of the above examples use temporal difference (TD) learning [9]. In the simplest form of TD learning an agent stores a scalar value for the estimated reward of each state, $V(s)$. The agent selects the action which leads to the state with the highest utility. In the tabular case, the value function is represented as an array indexed by states. At each decision point or step of an episode the agent updates the value function according to the observed rewards. TD uses bootstrapping, where an agent updates its state value with the reward received on the current time step and the difference between the current state value and the state value on the previous time step. $TD(\lambda)$ updates all previously encountered state values according to an exponential decay, which is determined by λ.

$$V(s^T) = \Re^T$$
$$V(s^i) = (1-\lambda)V(s^i) + \lambda V(s^{i+1})$$

$\Re^T$ is the final reward at the end of an episode. Due to space restrictions we refer the reader to Sutton and Barto [9] for a full description of the algorithm.

1.4 Function Approximation

Under certain technical conditions $TD(\lambda)$ is guaranteed to converge to an optimal solution in the tabular case, where a unique value can be stored for every state. But, many interesting problems such as Hearts have a restrictively large number states. Given a deck of 52 cards with each player being dealt 13 cards, there are $52!/13!^4 = 5.4 \times 10^{28}$ possible hands in the game. But, this is only a lower bound on actual size of the state space because it does not consider the possible states that we could encounter while playing the game. Regardless, we

cannot store this many states in memory, and even if we could, we do not have the time to visit and train in every state.

Therefore, we need some method to approximate the states of the game. In general this is done by choosing a set of features which approximate the states of the environment. A function approximator must map these features into a value function for each state in the environment. This generalizes learning over similar states and increases the speed of learning, but potentially introduces generalization error as the features will not represent the state space exactly.

One of the simplest function approximators is the linear perceptron. A perceptron computes a state value function as a weighted sum of its input features. In a learning task the perceptron only has to update the appropriate weights for the input features. In a state, s, we update the weights, w, according to the output error for a state, which is provided by TD learning:

$$w \leftarrow w + \alpha \cdot error \cdot \phi_s$$

where ϕ_s is the input features for the current state.

The output of a perceptron is a linear function of its input. Thus, a perceptron can only learn linearly separable functions. But most interesting problems require a nonlinear mapping from features to state values. In many cases one would turn to a more complex function approximator, such as a neural network or radial-basis functions. Neural networks can represent any nonlinear function, given several technical conditions on learning parameters and number of hidden layers [10]. Neural networks, however, suffer from slow learning rates and large computation costs, making them less than ideal in practice. So, instead of using a more complicated function approximator, we instead use a more complicated set of features with a simple perceptron.

This can be demonstrated with a simple example: Consider the task of learning the XOR function based on two inputs[1]. A two input perceptron cannot learn this function because XOR is not linearly separable. But, if we simply augment the network input with an extra input which is the logical AND of the first two inputs, the network can learn the optimal weights. In fact, all subsets of n points are always linearly separable in $n-1$ dimensional space. Thus, given enough features, a non-linear problem can have a exact linear solution.

2 Learning to Play Hearts

Before we look at Hearts, we briefly discuss the features of backgammon which make it an ideal domain for TD learning. For instance, in backgammon, pieces are always forced to move forward, except in the case of captures, so games cannot run forever. Also, the moves available to each player are randomly determined by a dice roll, which means that the learning player encounters different lines of play each game, unlike in a deterministic game like chess where the same game can be repeated. Thus, self-play has worked well in backgammon.

[1] The output of an XOR function is 1 iff both inputs differ, otherwise the output is 0.

Hearts has some similar properties. Each player makes exactly 13 moves in every game, which means we do not have to wait long to evaluate the rewards associated with play. Thus, we can quickly generate and play training games. Additionally, cards are dealt randomly, so, like in backgammon, players are forced to explore different lines of play in each game. Another useful characteristic of Hearts is that even a weak player occasionally gets good cards. Thus, we are guaranteed to see examples of good play.

One key difference between Hearts and backgammon is that the value of board positions in backgammon tend to be independent. In Hearts, however, the value of any card is relative to what other cards have been played. For instance, it is a strong move to play the 10♡ when the 2-9♡ have already been played. But, if they have not, it is a weak move. This complicates the features needed to play the game.

2.1 Feature Generation

Given that we are using a simple function approximator, we need a rich set of features. In the game of Hearts, and card games in general, there are many very simple features which are readily accessible. For instance, we can have a single binary feature for each card that each player holds in their hand and for each card that they have taken (e.g. the first feature is true if Player 1 has the 2♡, the second is true if Player 1 has the 3♡, etc.). This would be a total of 104 features for each player and 416 total features for all players. This set of features fully specifies a game, so in theory it should be sufficient for learning, given a suitably powerful function approximator.

However, consider a simple question like: "Does player 1 has the lowest heart left?" Answering this question based on such simple features is quite difficult, because the lowest heart is determined by the cards that have been played already. If we wanted to express this given the simple features described above, it would look something like: "[Player 1 has the 2♡] OR [Player 1 has the 3♡ AND Player 2 does not have the 2♡ AND Player 3 does not have the 2♡ AND Player 4 does not have the 2♡] OR [Player 1 has the 4♡ ...]" While this full combination *could* be learned by a sufficiently powerful function approximator, it is unlikely.

So, instead of using the most basic features possible in the game, we defined our own set of simple features, which we will call atomic features. These features are perfect-information features, so they depend on the cards other players hold. Then we built higher level features by combining these atomic features together. One set of atomic features used for learning can be found in Appendix A.

We might be able to sit down and write all useful combinations of our atomic features, but this would be tedious, error-prone, and time consuming. Instead, given a set of atomic features, we generated new features by taking successive pair-wise AND combinations of all atomic features. Obviously we could take this further by adding OR operations and negations. But, to limit feature growth we currently only consider the AND operator.

2.2 TD Parameters

For all experiments described here we used TD-learning as follows. The value of λ was set to 0.75. We first generated and played out a game of Hearts using a single learning player and either three expert search players [2], or three copies of our learned network for self-play. Moves were selected using the $\max^n$ algorithm with a lookahead depth of one to four, based on how many cards were left to play on the current trick. We used our own network as the evaluation function for our opponents.

To simplify the learned network we only evaluated the game in states where there were no cards on the current trick. Additionally we did not learn in states where all the points had already been played. After playing a game we computed the reward for the learning player and then stepped backwards through the game, using TD-learning to update our target output and train our perceptron to predict the target output at each step. We did not attempt to train using more complicated methods such as TDLeaf [11].

2.3 Learning To Avoid the Q♠

Our first learning task was to predict whether we would take the Q♠. We trained the perceptron to return an output between 0 and 13, the value of the Q♠ in the game. In practice, we cut the output off with a lower bound of $0 + \epsilon$ and an upper bound of $13 - \epsilon$ so that the search algorithm could always distinguish between states where we expected to take the Q♠ versus states where we already had taken the Q♠, preferring those where we had not yet taken the queen. The perceptron learning rate was set to $1/(13 \times number\ active\ features)$.

We used 60 basic features, listed in Appendix A, as the atomic features for the game. Then, we built pair-wise, three-wise and four-wise combinations of these atomic features. The pair-wise combinations of these features results in 1,830 total features, three-wise combinations of the atomic features results in 34,280 total features, and four-wise combinations of features results in 523,685 total features. But, many of these features are mutually exclusive and can never occur. We initialized the feature weights randomly between $\pm 1/numfeatures$.

The average score of the learning player during training is shown in Figure 1. This learning curve is performance relative to the expert program produced in [2]. Except for the four-wise features, we did five training runs of 200,000 games against the expert program. Scores were then averaged between each run and over a 5,000 game window. With 13 points in each hand, evenly matched players would expect to take 3.25 points per hand. The horizontal line in the figure is this break-even point between the learned and expert programs.

These results demonstrate that the atomic features alone are insufficient for learning to avoid taking the Q♠. The pair-wise features are also insufficient, but are better than the atomic features alone. Both the three-wise and four-wise combinations of features, however, are sufficient to beat the expert program. While the four-wise combinations of features are generally better than the three-wise combinations, each training game with the four-wise features is roughly ten times slower than with the three-wise features.

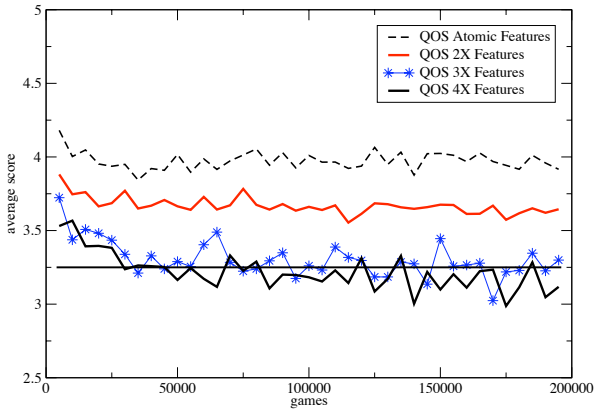

Fig. 1. Learning to not take the Q♠ using various combinations of atomic features. The break-even point is at 3.25

Table 1. Features predicting we can avoid the Q♠.

Rank	Weight	Top Features - Predicting we will *not* take the Q♠			
1.	-0.103	One of J-2♠		Lead	Q♠ player has no other ♠
2.	-0.097	One of J-2♠	No ♡	Lead	Q♠ player has no other ♠
3.	-0.096	Two of J-2♠	We have K♠		Q♠ player has two other ♠.
4.	-0.093	One of J-2♠	No ♣	Lead	Q♠ player has no other ♠
5.	-0.090	One of J-2♠	No ♢	Lead	Q♠ player has no other ♠
148.	-0.040	One of J-2♠	We have Q♠		Lead player has no ♠

We next analyze the important features learned by the player using four-wise combinations of features. Table 1 shows some of the features which best predict avoiding the Q♠. This player actually uses all atomic, pair-wise, three-wise and four-wise features, so some of the best features in this table only have three atomic features. Weights are negative because they reduce our expected score in the game.

There are a few things to note about these features. First, we can see that the highest weighted feature is also one learned quickly by novice players: If the player with the Q♠ has no other spades and we can lead spades, we are guaranteed not to take the Q♠. In this case, leading a spade will force the Q♠ onto another player.

Because we have generated all four-wise combinations of features, and this feature only requires three atomic features to specify, we end up getting the same atomic features repeated multiple times with an extra atomic features added. The features ranked 2, 4 and 5 are copies of the first feature with one extra

Table 2. Features predicting we will take the Q♠.

Rank	Weight	Top Features - Predicting we will take the Q♠			
1.	0.125	We have Q♠	One of J-2♠		Lead
2.	0.123	We have Q♠	One of J-2♠		
3.	0.117	We have Q♠	No ♣	No ♡	Lead
4.	0.116	Only A/K/Q♠			Lead
5.	0.112	We have Q♠	No ♣	No ♡	No ♢

atomic feature added. The 148th ranked feature should seem odd, but we will explain it when looking at the features which predict that we will take the Q♠.

The features that best predict taking the Q♠ are found in table 2. It takes a bit more work to understand these features. We might expect to find mirrored versions of the features from table 1 in table 2 (eg. we have a single Q♠ and the player to lead has spades). This feature is among the top 300 (out of over 500,000) features, but not in the top five features.

What is interesting about table 2 is the interactions between features. Again, we see that the atomic features which make up the 2nd ranked feature are a subset of the highest ranked feature. In fact, these two atomic features appear 259 times in the top 20,000 features. 92 times they are part of a feature which decreases the chance of us taking the Q♠, while 167 times they increase the likelihood. We can use this to explain what the perceptron has learned: Having the Q♠ with only one other spade in our hand means we are likely to take the Q♠ (feature 2 in table 2). If we also have the lead (feature 1 in table 2), we are even more likely to take the Q♠. But, if someone else has the lead, and they do not have spades (feature 148 in table 1), we are much less likely to take the Q♠.

In retrospect, the ability to do this analysis is one of the benefits of putting the complexity into the features instead of the function approximator. If we rely on a more complicated function approximator to properly learn weights, it is very difficult to analyze the resulting network. Because we have simple weights on more complicated features it is not difficult to analyze what has been learned.

2.4 Learning to Avoid Hearts

We used similar methods to predict how many hearts we would take within a game, and learned this independently of the Q♠. One important difference between the Q♠ and points taken from hearts is that the Q♠ is taken by one player all at once, while hearts are gradually taken throughout the course of the game. To handle this, we removed 14 Q♠ specific features and added 42 new atomic features to the 60 atomic features used for learning the Q♠. The new features were the number of points (0-13) taken by ourselves, the number of points taken (0-13) by all of our opponents combined, and the number of points (0-13) left to be taken in the game.

Given these atomic features, we then trained with all atomic (88), pairwise (3,916) and three-wise (109,824) combinations of features. As before, we

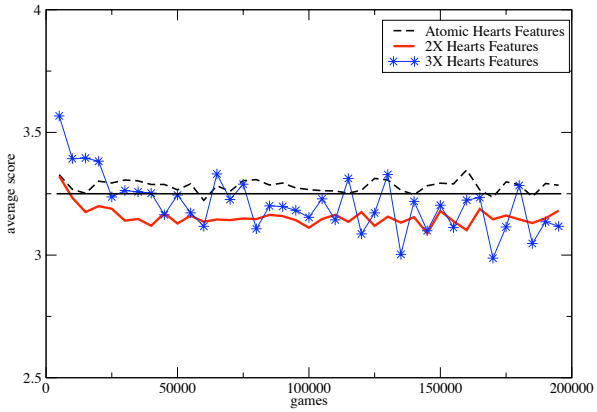

Fig. 2. Learning to not take the Hearts using various combinations of atomic features

present the results averaged over five training runs (200,000 games each) and then smoothed over a window of 5,000 games. The learning graph for this training is in Figure 2. An interesting feature of these curves is that, unlike when learning the Q♠, we begin to learn more slowly (per game) when we go from two-wise to three-wise features. It appears, then, that learning to avoid taking hearts is a bit easier than avoiding the Q♠. Because of this, we did not try all four-wise combinations of features.

2.5 Learning Both Hearts and the Q♠

With two programs that separately learned partial components of the full game of Hearts, the final task is to combine them together. We did this by extracting the most useful features learned in each separate task, and then combined them to learn to play the full game. The final learning program had the top 2,000 features from learning to avoid hearts and the top 10,000 features from learning to avoid the Q♠.

We tried training this program in two ways: first, by playing against our expert program, and second, by self-play. The first results are in Figure 3. Instead of plotting the learning curve, which is uninteresting for self-play, we instead plot the performance of the learned network against the expert program. We did this by playing games between the networks that were saved during training and the expert program. So, each data point is the average score per hand of the learned player after playing 1400 hands against the expert program. The horizontal line is the break-even point between the programs. Since there are 26 points in the full game, the break-even point falls at 6.5 points. Since lower scores are better,

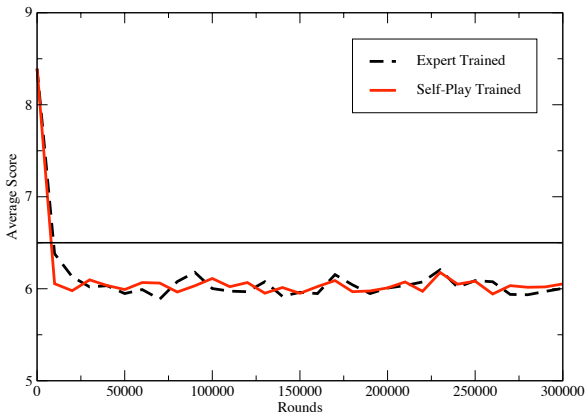

Fig. 3. Performance of self-trained player and expert-trained player against expert program

we see that both the self-trained player and the expert-trained player learn to beat the expert by the same rate, about 1 point per hand.

Then, in Figure 4 we show the results from taking corresponding networks trained by self-play and expert-play and playing them in tournaments against each other. Although both of these programs beat the previous expert program by the same margin, the program trained by self play managed to beat the expert-trained program by a large margin; again about 1 point per hand.

While we cannot provide a decisive explanation for why this occurs, we speculate that the player which only trains against the expert does not sufficiently explore the state space, and so does not learn to play well in certain situations of the game where the previous expert always makes mistakes. The program trained by self-play, then, is able to exploit this weakness.

3 Conclusions and Future Work

The work presented in this paper presents a significant step in learning to play the game of Hearts and in learning for multi-player games in general. But, there are many issues which still need to be investigated.

For instance, we did all of our learning in the perfect information variant of Hearts, relying on our ability to do Monte-Carlo search to handle imperfect information. It would be worthwhile to build a similar program that learned to play the imperfect information game, and then to compare the resulting programs to see which one was stronger.

Similarly, we used the $\max^n$ algorithm for move selection in the game. There are other algorithms that can be used including the paranoid algorithm [12] or

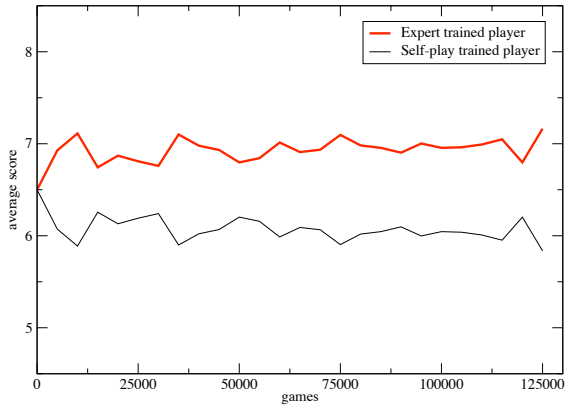

Fig. 4. Performance of self-trained player vs. expert-trained player

soft-max^n [13]. It would be useful to see if there are any advantages or disadvantages to using these algorithms during training and/or play.

Finally, we would like to eventually learn to play the full game of Hearts well, which includes passing cards between players and learning to prevent other players from shooting the moon. We expect that we can train the current program to effectively stop other players from shooting the moon, but it will be more of a challenge to decide when the program should shift its strategy to trying to shoot the moon.

Acknowledgements

We would like to thank Rich Sutton for his feedback and suggestions regarding this work. We would also like to thank Mark Ring for many useful discussions on various learning approaches. This work was supported by Alberta's Informatics Circle of Research Excellence (iCORE).

A Atomic Features Used to Learn the Q♠

Unless explicitly stated, all features refer to cards in our hand. The phrase "to start" refers to the initial cards dealt. "Exit" means we have a card guaranteed not to take a trick. "Short" means we have no cards in a suit. "Backers" are the J-2♠. "Leader" and "Q♠ player" refers to another player besides ourself.

we have Q♠	we have A♠	we have K♠
$\geq$5 spades besides AKQ♠	0 spades besides AKQ♠	1 spades besides AKQ♠
2 spades besides AKQ♠	3 spades besides AKQ♠	4 spades besides AKQ♠
$\geq$ 3 diamonds to start	0 diamonds to start	1 diamonds to start

2 diamonds to start	currently short diamonds	currently not short diamonds
opponent short diamonds	exit in diamonds	$\geq$ 3 clubs to start
0 clubs to start	1 clubs to start	2 clubs to start
currently short clubs	currently not short clubs	opponent short clubs
exit in clubs	$\geq$ 3 hearts to start	0 hearts to start
1 hearts to start	2 hearts to start	currently short hearts
currently not short hearts	opponent short hearts	exit in hearts
we have single Q♠	we have single A♠	we have single K♠
we have lead	Q♠ player has 0 backers	Q♠ player has 1 backers
Q♠ player has 2 backers	Q♠ player has $\geq$3 backers	Q♠ player has 0 shorts
Q♠ player has 1 shorts	Q♠ player has 2 shorts	Q♠ player has 3 shorts
Q♠ player has short diamonds	Q♠ player has short clubs	Q♠ player has short hearts
leader short spades	leader short diamonds	leader short clubs
leader short hearts	leader not short spades	leader not short diamonds
leader not short clubs	leader not short hearts	we have forced high spade
we have forced high diamond	we have forced high club	we have forced high heart

References

1. Ginsberg, M.: Gib: Imperfect information in a computationally challenging game (2001)
2. Sturtevant, N.R.: Multi-Player Games: Algorithms and Approaches. PhD thesis, Computer Science Department, UCLA (2003)
3. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall; 2nd edition, Englewood Cliffs, NJ (2002)
4. Luckhardt, C., Irani, K.: An algorithmic solution of N-person games. In: AAAI-86. Volume 1. (1986) 158–162
5. Perkins, T.: Two search techniques for imperfect information games and application to hearts. University of Massachusetts Technical Report **98-71** (1998)
6. Fujita, H., Ishii, S.: Model-based reinforcement learning for partially observable games with sampling-based state estimation. In: Advances in Neural Information Processing Systems, Workshop on Game Theory, Machine Learning and Reasoning under Uncertainty. (2005)
7. Fürnkranz, J., Pfahringer, B., Kaindl, H., Kramer, S.: Learning to use operational advice. In: Proceedings of the 14th European Conference on Artificial Intelligence. (2000)
8. Stone, P., Sutton, R.S.: Scaling reinforcement learning toward RoboCup soccer. In: Proc. 18th International Conf. on Machine Learning, Morgan Kaufmann, San Francisco, CA (2001) 537–544
9. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press (1998)
10. Mitchell, T.: Machine Learning. McGtaw-Hill (1997)
11. Baxter, J., Trigdell, A., Weaver, L.: Knightcap: a chess program that learns by combining TD(λ) with game-tree search. In: Proc. 15th International Conf. on Machine Learning, Morgan Kaufmann, San Francisco, CA (1998) 28–36
12. Sturtevant, N.R., Korf, R.E.: On pruning techniques for multi-player games. In: AAAI-2000. (2000)
13. Sturtevant, N.R., Bowling, M.H.: Robust game play against unknown opponents. In: To Appear, AAMAS-2006. (2006)
14. Tesauro, G.: Temporal difference learning and td-gammon. Communications of the ACM **38 (3)** (1995) 58–68