Learning to Use Operational Advice

Johannes Fürnkranz[1] and **Bernhard Pfahringer**[2] and **Hermann Kaindl**[3] and **Stefan Kramer**[4]

Abstract.
We address the problem of advice-taking in a given domain, in particular for building a game-playing program. Our approach to solving it strives for the application of machine learning techniques throughout, i.e., for avoiding knowledge elicitation by any other means as much as possible. In particular, we build upon existing work on the operationalization of advice by machine and assume that advice is already available in operational form. The relative importance of this advice is, however, not yet known and can therefore not be utilized well by a program. This paper presents an approach to determine the relative importance for a given situation through reinforcement learning. We implemented this approach for the game of Hearts and gathered some empirical evidence on its usefulness through experiments. The results show that the programs built according to our approach learned to make good use of the given operational advice.

1 INTRODUCTION

One of the major problems of building "intelligent" machines is to make knowledge of the given domain available for their use. For human apprentices, it is often sufficient to provide high-level advice. In contrast to humans, however, it is much more difficult for machines to operationalize such advice. Even if it is (made) operational, the problem remains of how to determine the relative importance of such pieces of advice for a given situation. That is, how can a machine make use of operational advice?

We focus on this problem in the context of building a game-playing program. In order to make the role of given advice more transparent, we chose the game Hearts.[5] Our approach does not make use of deep searches. Apart from programming the rules of a given game, we distinguish two subtasks involved in building such a game-playing program:

1. acquiring important knowledge for playing the game;
2. determining the relative importance of the pieces of knowledge for a given state of a game.

Subtask 1 is mostly dealt with by hand-crafting components of an evaluation function or features of a neural net, subtask 2 by tuning parameters in the sense of relative weights either manually or through some form of machine learning. However, the overall problem is rarely, if ever, addressed completely through machine learning.

In this paper, we build on previous work in machine learning for *both* subtasks and show how it can be combined successfully. For subtask 1 we take the results of FOO (First Operational Operationalizer) [9, 10] as given: advice for the game of Hearts, transformed into operational form. In the remainder of the paper, we simply use the terms "advice" or (interchangeably) "heuristics" to refer to advice in operational form.

Note, however, that such operational advice does not include information about how to combine and relate pieces of advice to each other. In particular, some of them compute important properties of the game state, but no information is given about how to use these properties for selecting a good move. Other pieces of advice suggest certain subsets of the legal moves, but no information is given about when it is reasonable to follow these suggestions and what to do with conflicting and/or overlapping suggestions.

We are not aware of any game-playing program that made use of this operational advice for really playing Hearts. It is not immediately obvious how a program should make a non-random selection from the proposed set of cards, which may even comprise the complete set of legal moves to be played. We address this issue by determining the relative importance of the various pieces of advice for a given state of a game, i.e., viewing it as subtask 2 above. This subtask is dealt with in this paper through reinforcement learning.

This paper is organized in the following manner. First, we have a closer look at the operational advice given from FOO. Then we explain our approach to learning for making good use of such operational advice. In order to provide some evidence for its usefulness, we present experimental results. Finally, we discuss our approach more generally and briefly survey related work.

2 A CLOSER LOOK AT THE AVAILABLE KNOWLEDGE

In Appendix D of [9], operationalizations of 11 pieces of advice for the game of Hearts are derived. We reuse this knowledge for demonstrating the feasibility of our approach. In order to make it easier to understand this approach for learning how to make good use of such advice, let us have a closer look at this available knowledge first. In Appendix A we describe the individual pieces of advice in more detail.

We distinguish two classes of advice:

1. *Advice for move selection*
   Each of these pieces of advice suggests moves to be selected in the sense of (a set of) cards to be played next. A simple example is "Get the Lead", while the much more intricate "Avoid Taking Points (Search)" is in its operational form a heuristic search procedure (its derivation is also explained in [10]).

---

[1] Austrian Research Institute for Artificial Intelligence, Schottengasse 3, A-1010 Wien, Austria, email: juffi@ai.univie.ac.at
[2] Department of Computer Science, University of Waikato, Hamilton, New Zealand, email: bernhard@cs.waikato.ac.nz
[3] Siemens AG Österreich, Geusaugasse 17, A-1030 Wien, Austria, email: hermann.kaindl@siemens.at
[4] Machine Learning and Natural Language Processing Lab, Albert-Ludwigs-Universität Freiburg, D-79110 Freiburg im Breisgau, Germany, email: skramer@informatik.uni-freiburg.de
[5] For a brief sketch of the rules of the game see, e.g., http://nelson.oit.unc.edu/~alanh/hearts.html. In the experiments reported in this paper, we followed the variant that is laid out in Appendix A.1 of [9] (3 players, shooting-the-moon, jack-of-diamonds, no passing of cards).

2. *Advice for state abstraction*
   Each of these pieces of advice makes a certain abstraction of the current game state. They use information of the given state as known, such as the cards of the player's own hand as well as information of the past history of the game so far, in order to make predictions about the cards of the other players' hands. A simple example that uses both kinds of information is "Queen Out".

Both classes of advice are useful for playing the game of Hearts well and, in fact, both are involved in our learning approach presented in this paper. However, it should be clear that for playing this game really well, more pieces of advice would be needed than those reused here. For instance, the first of these classes lacks advice for move selection to cope with important aspects like taking the Jack of diamonds and "shooting the moon". The second class abstracts from some important properties of the current state of the game, e.g., the player's cards. Still, we found this given advice useful and sufficient as a basis for our experiment. Moreover, it is available to the public, which facilitates reproducibility.

It is also important for understanding our approach to reflect more closely on certain properties of the class of advice for move selection. Applying those pieces of advice in a given situation may in general propose several moves, sometimes even all of the legal moves. The sets of cards suggested for being played by the various pieces of advice can be disjoint, which means a conflict of the move suggestions. In general, these sets are not necessarily disjoint but will overlap, and the cardinality of their intersection is typically greater than one. In addition, one set may subsume another set, i.e., one piece of advice is more general than another, more specific one. In summary, the class of advice for move selection should be interpreted as a "plausible move generator" rather than a means for making a clear decision for a single "best" move.

3 LEARNING THE RELATIVE IMPORTANCE OF ADVICE

As discussed in the previous section, there are two different types of advice: advice for state abstraction and advice for move selection. We have to address the following problems:

- How do we integrate the two different types of advice?
- How do we deal with conflicting and overlapping advice?

We address both problems by learning a function that maps abstracted states to weights of the move selection heuristics (see Figure 1). So, the task of the learning algorithm is to learn the relative importance of the different move selection heuristics in the current states, which are represented in an abstract way. Conventional reinforcement learning techniques learn a value function that maps state representations to action values. In our approach, a value function is learned which maps the abstract state descriptions produced by the operational state abstraction heuristics to weights for the abstract move selection heuristics. These actions are "abstract" in the sense that they are not directly moves to be played, but rather heuristics for move selection. This is novel in reinforcement learning, where usually "abstract actions" refers to temporally abstract actions.

When the learner considers a move, it adds up the weights of all heuristics that suggest this particular move (possibly among other alternatives). Among the moves that have the highest cumulative weight, one is chosen randomly (with equal probability). Note that although these learned weights can be interpreted as expected future rewards (see below), we found that maximizing this expected reward

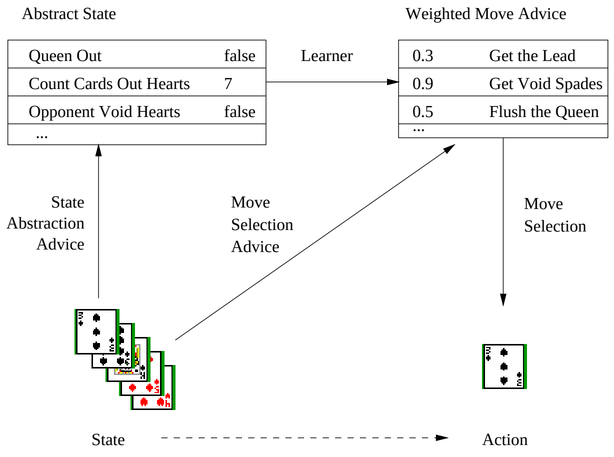

Figure 1. Our learning architecture.

by following the advisor with the highest weight did actually hurt performance. We believe that the reason for this is that weights of the individual advisors cannot be treated independently, which is a topic for further work.

Before learning, all weights are initialized equally and the system plays like the VOTING benchmark player (see section 4). This facilitates the evaluation of the learning progress.

We experimented with two straightforward representations for the value function: a simple look-up table and a neural network. In both cases, we trained the learner using a simple TD(0) reinforcement learning approach.

The adjustment of estimates of the reward works as follows: after each trick, the current estimates are used to adjust the estimates of those heuristics that proposed the chosen move in the previous state. At the end of each game, the final reward is simply the total number of points that the learning player has accumulated in that game. This final reward is mapped into the interval $[0, 1]$, where 1 reflects the best possible outcome and 0 the worst. The weights of the move selection heuristics that did not suggest the last move played are left unchanged. Eventually, with this training method, the weights should converge to the expected reward of the corresponding move selection advice in the current situation, which is represented through the values of the state abstraction heuristics.

The look-up-table learner LEARNER-T does not use all available state abstraction heuristics: the card-counting heuristics (Count Cards Out, see Appendix A) are omitted because they would lead to a huge number of different abstract states. Furthermore, the eight opponent-void (behavior/past) heuristics have been combined with the corresponding four opponent-void (distribution) heuristics via a logical 'or' (see Appendix A). The resulting eight heuristics plus the Queen-Out advice are all Boolean, yielding only 512 abstract states. For each state the expected utility of each of the nine move selection heuristics is estimated separately. Thus a total of 4,608 numeric estimates define the mapping of abstract states to abstract actions.

For the temporal updates of the weights in the table we used the following simple undiscounted temporal difference rule:

$$w_i^t \leftarrow w_i^t + \alpha(w_i^{t+1} - w_i^t),$$

where w_i^t denotes the value of the i-th weight at time t. We arbitrarily picked the value 0.1 for the step-size parameter α [14].

The design of the neural-net learner LEARNER-N is as follows. We train a neural network with seventeen input units (each represent-

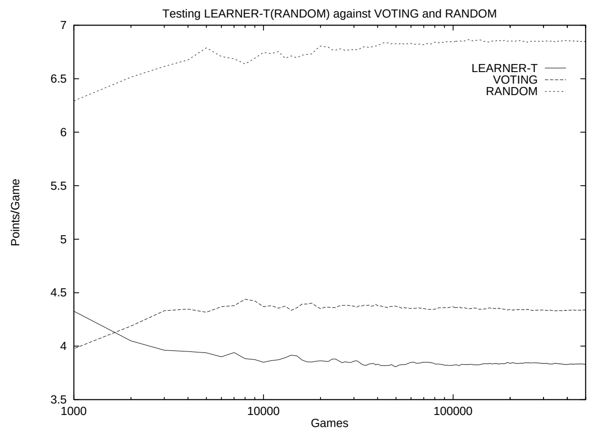

Figure 2. Results of LEARNER-T (trained against two RANDOM players) against RANDOM and VOTING players.

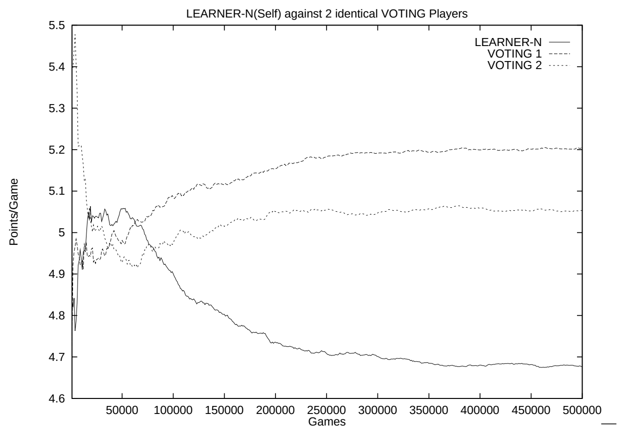

Figure 3. Test results of LEARNER-N (trained against two independently learning copies of itself) against two VOTING players.

ing one state abstraction advice) and nine output units (each representing one move selection advice). Thirteen of the input units are Boolean (encoded as 0 and 1) and four of them encoded integers in the range of [0,13], which are linearly mapped to [0,1]. Input and output layers are fully connected to a hidden layer with thirteen units. After each trick, the network learns via one step of backpropagation. The weights w_i^t, which were predicted for each move selection advice at the current trick, are adjusted towards the weights w_i^{t+1}, the predictions for the next trick (or the final result of the game, in the case of the last trick). This is realized by using the weights w_i^{t+1} as training signals for the weights w_i^t.

For implementing this network we used Jude Shavlik's and Ray Mooney's publicly available LISP-code and relied on the default paramenters provided therein (see `http://www.cs.utexas.edu/users/ml/ml-progs.html`). We did not make any attempts to optimize the learning parameters or the network architecture, but relied on the default settings, which we believe were sufficient to demonstrate the validity of our approach.

4 EXPERIMENTAL RESULTS

We made several experiments with both LEARNER-T and LEARNER-N. For both learning and performance comparison, we needed other Hearts playing programs as well:

- a RANDOM player that plays randomly in the sense that one of the legal moves is chosen with equal probability;
- a VOTING player that directly uses the operational move selection advice that we reuse in our approach — each advice votes for all cards it suggests and among the cards receiving the highest number of votes, one is selected randomly with equal probabilities.

Whereas the design of the former is straightforward, that of the latter is the best that makes use of the given operational advice without any learning and without introducing any additional domain knowledge. In particular, it was not clear how it could utilize the advice for state abstraction. It should also be noted, that the behavior of the VOTING player is identical to that of a learner which is initialized with identical values for all target weights.

Since we wanted to monitor progress during learning, we interleaved learning and testing the performance: runs of 1,000 learning games alternated with runs of 1,000 testing games against different opponents. For these testing games the parameters of the respective learner were "frozen" and learning was turned off.

Figure 2 shows test results of LEARNER-T(RANDOM), an instance of LEARNER-T that is trained by playing against two RANDOM players, tested by playing against RANDOM and VOTING. The x-axis of the graph denotes the logarithm of the number of training games played, while the y-axis denotes the cumulative average of the points scored by each player per game on the same number of test games.

After 1,000 games, the performance of LEARNER-T is below that of VOTING (i.e., it gets more points), although both clearly outperform RANDOM. In the next several thousand games, the performance of LEARNER-T increases steadily at the expense of the respective performances of RANDOM and VOTING. After about 10,000 games, however, the performance peaks. The corresponding results of LEARNER-N are similar. Much as expected, however, its learning rate was slower, and its peak performance was slightly better than that of LEARNER-T.

In addition, we performed self-training experiments: three instances of a learner, which were initialized with different random weights, learned independently by playing against each other. Figure 4 shows the results of one such experiment with LEARNER-N, where we arbitrarily picked one of the three learners and monitored its progress by testing it against two VOTING players. The axes are the same as in the previous figure, except that the games are plotted on a non-logarithmic scale because the variance for the first 50,000 games is considerably higher in experiments with LEARNER-N.

While the three players are about equal in the beginning, the learner steadily improves until it has reached a certain maximum performance. It is interesting to note that although the learner is evaluated against two identical VOTING players, it apparently has learned to exploit certain characteristic mistakes that depend on its position relative to the learner: the VOTING player that plays after the learner performs substantially worse than the VOTING player that sits before the learner.[6]

Table 1 shows the results of three tournaments of 100,000 games that were played with each of the three independent instances of

[6] Note that this effect is not due to an absolute order of the players, because the player that opens a game is chosen randomly in the beginning (whoever has the lowest club card), the winner of the previous trick opening the next trick. Instead, it merely depends on the relative order of the players to each other: it can make a significant difference whether a good player sits immediately before or behind a bad player.

Table 1. Tournament results of three instances of LEARNER-N trained by playing against each other. Each played in a tournament against VOTING and RANDOM. The lines of the table show the average number of points per game that were scored by the three participants.

i	LEARNER-N-i	VOTING	RANDOM
1	3.86	4.17	6.90
2	3.45	4.36	7.22
3	3.59	4.26	6.97

LEARNER-N (with frozen weights) resulting from the self-training experiment against a VOTING and a RANDOM test partner. The two opponents changed their seats after 50,000 games, so that the order effect discussed in footnote 2 averages out. All of our learners consistently achieved better results than their opponents.

Again, the results for LEARNER-N and LEARNER-T are similar. A comparison of their respective results shows that LEARNER-T reaches its performance peak much earlier than LEARNER-N. This is mostly due to the simpler learning scheme and the smaller state space of LEARNER-T. Interestingly, in some of the experiments with LEARNER-N (although in none of the experiments with LEARNER-T) we noted a small but systematic degradation in performance after the optimum has been reached. We do not have an explanation for this over-training phenomenon yet.

We have also seen evidence that LEARNER-N can learn faster from a RANDOM training partner than from a VOTING training partner or from self-play. We interpret this in the way that self-training (and even more so learning from VOTING) seems to be affected by the exploration vs. exploitation trade-off.

In summary, our experiments provide some empirical evidence that our approach works. Both through simple table-based and default neural-net learning, it was possible to clearly improve on the performance of an otherwise identical player that does not know about the relative importance of the given advice. Apparently, the learners came up with reasonable weights, that allow making good use of the advice.

5 DISCUSSION

It should be noted that we did not (yet) attempt to build a strong Hearts playing program. In order to make the point, we only used straightforward reinforcement learning approaches. We restricted ourselves to TD(0) learning, and made no attempts to optimize the parameter settings of the learning algorithms we used.

Still, in order to make reinforcement learning feasible in our new framework that resulted from integrating state abstraction and move selection advice into the learning process, we came up with the novel approach of learning weights for abstract states and their relation to abstract actions in the sense of move selection heuristics.

Similarly, we made no attempt to improve the performance by providing additional pieces of advice. We chose this particular subset of useful advice for the game of Hearts in order to be consistent with the previous work [9] that we build upon. However, some aspects of the game (like capturing the Jack of diamonds or shooting the moon) are not explicitly addressed by these heuristics (although taken into account in the evaluation). We believe that the fact that we achieved almost identical performance peaks in various learning settings, can be interpreted as evidence that the provided heuristic knowledge allows only for a certain amount of improvement.

Regarding the experimental design, more sophisticated variants could be chosen. In particular, variation could be added by learning against various partners as suggested in [3]. Despite some amount of randomness in the game due to the dealing of cards in the beginning, the playing strategies are deterministic. Thus, such a mixed training approach may be a better choice than self-training [3, 11] and training against random players.

A challenging research problem is to find a solution for the reinforcement learning training of multiple advisors. In principle, each weight of a move selection advice predicts the expected reward for applying this operator in the current situation. Hence, the most reasonable playing strategy should be to pick the operator that promises the maximum reward. However, due to the inherent uncertainty in these estimates, we believe that a strategy which takes into account the advice of more than one advisor should be preferable to one that always follows the advisor with the maximum weight. Although preliminary experiments, which showed that voting actually performed better than maximizing, confirmed us in this belief, this playing strategy is not yet reflected in the learning procedure, which still learns the expected reward for each advisor independently.

6 RELATED WORK

Ever since Samuel's seminal work [12, 13], machine learning was applied to building game-playing programs. Due to lack of space, we cannot give a comprehensive overview here, but for computer chess, e.g., let us refer to [5]. Reinforcement learning [14] — already present in Samuel's program — is one of the main techniques for learning the weights of evaluation functions. TD-Gammon, a very strong Backgammon program trained by reinforcement learning [15, 16] is one of the major success stories in this area. Naturally, this approach has been tried for other games, including Hearts [7].

Samuel [12] has already noted that the main deficiency of such approaches is their dependence on carefully selected features. While other authors tried to automatically construct new features from a few basic features [1, 17], advice-taking may be viewed as an attempt to solve this problem by human-machine collaboration. Mostow's work on operationalizing human advice [9, 10] is one approach into this direction. In this context, we view our contribution as an automated approach for integrating different, contradicting pieces of advice into a coherent playing strategy.

There have been several other approaches with similar goals. For one, the HOYLE game-playing system [4] consists of a variety of general, game-independent advisors, whose utility for a particular game is adapted by learning techniques. Like in our approach, the different pieces of advice are combined by voting. The main differences to our approach are that the weights are learned by supervised learning [2] and that each advisor can only vote for or against a single move.

In a non-game playing setting, Maclin & Shavlik [8] assume an external observer who provides advice (in addition to the external reward signal) to the learner. Technically, they compile advice first represented as rule-sets into additional hidden units of a neural network using the KBANN (Knowledge-Based Artificial Neural Networks) approach. They do not address the issue of overlapping or conflicting advice.

Gordon & Subramanian [6] created a system that first operationalizes high-level advice into rules connecting specific states with primitive actions. A second phase employing a genetic algorithm for reinforcement learning further refines these rule sets. Refinements include the determination of appropriate rule strengths as well as symbolic modifications of the rule set. Even though the first part might seem similar to Mostow's work that we build upon, Mostow's notion of advice is much broader including advice on both good actions (move-selection) and useful state descriptions (state-abstraction).

In summary, our suggested architecture differs from other approaches by its separation of state abstraction and move selection advice, and the learning framework that proposes to relate one to the other. Moreover, while several of the previous approaches use reinforcement learning for refining advice, our approach learns the relative weight of pieces of advice in order to address the issue of overlapping or conflicting advice.

7 CONCLUSION

In summary, this work shows a complete alternative to hand-crafting evaluation functions through utilizing (known) machine learning techniques for making given advice useful:

1. making the advice operational through learning (reused from [9, 10]), instead of "manual" knowledge acquisition;
2. making use of this operational advice by automatically determining the relative importance of the given pieces of advice for a given situation through reinforcement learning.

The major contribution of this paper is an approach that makes this combination feasible. Instead of learning parameters of an evaluation function, we let the machine learn the relative importance of given advice for proposing moves. Our experimental data confirm that the resulting programs defeat both a random player and a player that tries to use (the available) operational knowledge directly (through voting, i.e., without extra knowledge about the relative importance).

Using our approach, a game-playing program can be built by just implementing the rules and providing advice. Assuming that the program can make this advice operational, it also utilizes it well in the sense that it automatically learns quantitative knowledge about the relative importance of several pieces of advice. In this sense, it indeed learns to make use of operational advice.

A THE OPERATIONAL ADVICE REUSED

The operational advice reused from [9] can be intuitively paraphrased in natural language as follows.

If applicable, each *move selection advice* selects a subset of the legal moves that it suggests to be played in the current state.

Avoid Taking Points (Search): Avoid to take points during the current trick. This heuristic is implemented by a search procedure suggesting cards that will definitely avoid taking points during the current trick.

Flush the Queen: As long as the Queen of spades is not out, open with spades.

Safely Flush the Queen: As long as the Queen of spades is not out, open with a spade that is lower than the Queen.

Avoid Taking Points (Low card): If the current trick has points, play a card that is lower than the current highest card.

Get the Lead: Play a card that takes the current trick.

Get Void: Try to get rid of all cards of a suit. As the advice does not specify which suit, we expanded this heuristic into four pieces of advice, one for each of the four suits. Note that each of these heuristics suggests all legal moves in this suit whenever the player opens the round or is void in the suit led.

Each *state abstraction advice* computes some potentially useful property of the current game state.

Queen Out: Decide whether the Queen is in your cards or already out.

Opponent Void (Distribution): Decide whether your opponents are void in a certain suit by computing whether all cards of a given suit are out or in the player's hand. Again, we encode this with four pieces of advice, one for each suit.

Opponent Void (Behavior/Past): Decide whether an opponent is void in a given suit because he has in a previous round not followed this suit. This set of eight heuristics (four suits for two opponents) comprises two of Mostow's original heuristics.

Count Cards Out: For each suit, count the number of cards that are already out in this suit. This information is encoded as four pieces of advice with a value range of [0,13] each.

ACKNOWLEDGEMENTS

The Austrian Research Institute for Artificial Intelligence is supported by the Federal Ministry of Education, Science and Culture. We would like to thank Ray Mooney for making his Common Lisp ML library available to the public.

REFERENCES

[1] M. Buro, 'From simple features to sophisticated evaluation functions', in *Proceedings of the First International Conference on Computers and Games (CG-98)*, eds., J. van den Herik and H. Iida, p. 126, Springer-Verlag, (1998).

[2] S.L. Epstein, 'Identifying the right reasons: Learning to filter decision makers', in *Proceedings of the AAAI Fall Symposium on Relevance*, eds., R. Greiner and D. Subramanian, pp. 68–71. AAAI Press, (1994). Technical Report FS-94-02.

[3] S.L. Epstein, 'Toward an ideal trainer', *Machine Learning*, **15**, 251–277, (1994).

[4] S.L. Epstein, J. Gelfand, and J. Lesniak, 'Pattern-based learning and spatially-oriented concept formation in a multi-agent, decision-making expert', *Computational Intelligence*, **12**(1), 199–221, (1996).

[5] J. Fürnkranz, 'Machine learning in computer chess: The next generation', *International Computer Chess Association Journal*, **19**(3), 147–160, (September 1996).

[6] D. Gordon and D. Subramanian, 'A multistrategy learning scheme for agent knowledge acquisition', *Informatica*, **17**, 331–344, (1994).

[7] L. Kuvayev, 'Learning to play Hearts', in *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, p. 837, Providence, RI, (1997). AAAI Press. Extended Abstract, a longer version can be retrieved from `ftp://ftp.cs.umass.edu/pub/anw/pub/kuvayev/hearts-aaai97.ps`.

[8] R. Maclin and J.W. Shavlik, 'Creating advice-taking reinforcement learners', *Machine Learning*, **22**, 251–282, (1996).

[9] D.J. Mostow, *Mechanical Transformation of Task Heuristics into Operational Procedures*, Ph.D. dissertation, Carnegie Mellon University, Department of Computer Science, 1981.

[10] D.J. Mostow, 'Machine transformation of advice into a heuristic search procedure', in *Machine Learning: An Artificial Intelligence Approach*, pp. 367–403. Morgan Kaufmann, (1983).

[11] J.B Pollack and A.D. Blair, 'Co-evolution in the successful learning of backgammon strategy', *Machine Learning*, **32**(3), 225–240, (1998).

[12] A.L. Samuel, 'Some studies in machine learning using the game of checkers', *IBM Journal of Research and Development*, **3**(3), 211–229, (1959).

[13] A.L. Samuel, 'Some studies in machine learning using the game of checkers. ii - recent progress', *IBM Journal of Research and Development*, **11**(6), 601–617, (1967).

[14] R.S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

[15] G. Tesauro, 'Practical issues in temporal difference learning', *Machine Learning*, **8**, 257–278, (1992).

[16] G. Tesauro, 'Temporal difference learning and TD-Gammon', *Communications of the ACM*, **38**(3), 58–68, (March 1995).

[17] P.E. Utgoff and D. Precup, 'Constructive function approximation', in *Feature Extraction, Construction and Selection: A Data Mining Perspective*, eds., H. Liu and H. Motoda, Kluwer Academic Publishers, (1998).