MACHINE LEARNING IN COMPUTER CHESS: THE NEXT GENERATION

Johannes Fürnkranz*

1 INTRODUCTION

Ten years ago the *ICCA Journal* published an overview of machine learning approaches to computer chess (Skiena 1986). The author's results were rather pessimistic. In particular he concludes that

> "...with the exception of rote learning in the opening book few results have trickled into competitive programs."

and that

> "There appear no research projects on the horizon which offer reason for optimism."

In this paper we will update Skiena's work with research that has been conducted in this area since the publication of his paper. By doing so we hope to show that at least Skiena's second conclusion is no longer valid.

2 SUMMARY OF (Skiena 1986)

Skiena starts his overview by giving a motivation for machine learning research in the area of computer chess. Psychological studies, most notably (deGroot 1965), have shown that the differences in playing strengths between experts and novices are not so much due to differences in the ability to calculate long move sequences, but to which moves they start to calculate. For this preselection of moves chess players make use of chess patterns and accompanying promising moves and plans. Simon and Gilmartin (1973) estimate the number of a chess expert's patterns to be of the order of 10,000 to 100,000. Although the best programs nowadays can play on a grandmaster level, their architecture has nothing in common with the way humans play chess. It is still a long way until pattern-based computer chess programs will be competitive with brute-force programs (if they ever will), but their investigation may at least shed more light on the human cognitive processes that underly expert chess play.

Skiena then starts his tour through machine learning research in computer chess with rote learning approaches, in particular the use of opening books. Contrary to a quotation by Thompson and Condon implying that the opening book does not have a significant effect on improving playing strength it is nowadays undisputed that the opening preparation is at least as important for competitive programs as it is for human experts. As another application of rote learning Skiena mentions the research on the construction of perfect endgame databases which was still in its infancy at that time. The success of Ken Thompson's impressive work on 5-men endgame databases, which are now publicly available on 3 CD-ROMs, has shown both, the enormous magnitude of the task and its importance for chess. The latter point is illustrated by John Nunn's work on using these databases for a better understanding of endgames (Nunn 1994a) which resulted

*Austrian Research Institute for Artificial Intelligence, Schottengasse 3, A-1010 Wien, Austria, E-mail: juffi@ai.univie.ac.at, WWW: http://www.ai.univie.ac.at/ juffi

in two widely acknowledged endgame books (Nunn 1992; Nunn 1994b). However, it is questionable whether the construction of opening books and endgame databases can be considered as learning. We will not further discuss research in these areas.

Next, Skiena (1986) discusses work on advice taking chess programs. Here again the borderline between learning and programming is not so clear. His only example was the program of Zobrist and Carlson (1973) which was tutored by a chess master. However, the process of tutoring rather resembled programming in a high-level chess programming language than autonomous learning. Although an effort for developing a highly-flexible programming language for chess is highly desirable, we do not consider it as an endeavor in machine learning. A short bibliography of more recent approaches to advice taking chess programs can be found in (Michie and Bratko 1991). Most of them are confined to the endgame, the most notable exception being (George and Schaeffer 1990).

A significant body of research has gone into the task of learning to classify positions from a given chess endgame as wins or no-wins. Michalski and Negri (1977) and Shapiro and Niblett (1982) started research in this area by trying to learn correct classification rules for the KPK endgame. Since then, endgame databases for KPK, KPa7KR, or KRKN have become standard benchmark problems for induction algorithms. Research in this area is also important for computer chess, because if endgame positions could be correctly classified with a few rules, this would save many resources compared to the alternative of storing all positions of a variety of endgames. Besides, many of the endgame databases that are now available are not thoroughly understood by human experts. The most famous example are the attempts of grandmasters to defeat a perfect KQKR database. We will discuss further research in this area in section 3.

In section 4 we will discuss the use of *explanation-based learning* for computer chess. This technique was still in its infancy in 1986, which presumably is the reason why Skiena did not mention any research in this area in his overview.

In the area of learning by analogy Skiena reports no significant research. Since then this field of research has more or less been subsumed by *case-based reasoning*, which prospered in the second half of the 80s. We will discuss some attempts to apply these techniques to computer chess in section 5.

Learning algorithms for the automatic tuning of evaluation functions were only mentioned in one paragraph in (Skiena 1986). Nitsche (1982) describes how a least squares fitting technique can be used to determine appropriate weights for a given set of features using a collection of master games, by playing an opponent, or by self-play. Similar approaches will be discussed in section 6.

3 INDUCTION OF ENDGAME CLASSIFIERS

The induction of chess concepts is probably the task in computer chess that has been most intensively studied in machine learning research. Typically positions from a certain chess endgame are preclassified into the categories won or not-won. An inductive learning algorithm uses these positions to infer classification rules that are able to rederive the original classification of the positions. Thus only the rules have to be stored, which usually takes significantly less space than storing the entire look-up table for all positions. Besides the learned rules may also yield insight into poorly understood endgames and thus increase our knowledge of the game.

Very often only a subset of the positions — the so-called *training set* — is used for inducing the classification rules. The rest of the available positions (the *test set*) is then classified with the learned rules and for each of them the predicted result is compared with the true classification. The average accuracy of the rules for predicting the true result of the positions in the test set can be used as an estimate for the quality of the rules.

A major problem in this line of research, however, is to find a representation for the chess positions that allows the learner to induce meaningful rules. Typical inductive machine learning algorithms need a so-called *attribute-value representation* for the training instances. Each training example is specified with exactly one value for a fixed set of pre-defined attributes. The learned concept description is limited to tests for certain values of the given attributes. Thus if the positions are only represented with obvious attributes like the location of the pieces and the

right to move, the programs are not capable of making useful generalizations. Usually potentially useful information like the distance of certain pieces and useful patterns on the board (like kings' opposition) also have to be provided as attributes that can be used by the learning system.

Quinlan (1983) describes several experiments for learning rules for the KRKN endgame. More specifically he used his decision tree learning algorithm ID3 to discover recognition rules for positions of the KRKN endgame that are `lost-in-2-ply` and `lost-in-3-ply` respectively. From less than 10% of the possible KRKN positions ID3 was able to derive a tree that committed only 2 errors on a test set of 10,000 randomly chosen positions. However, Quinlan notes that this achievement was only possible through the right choice of the attributes that were used to represent the positions. Finding the right set of attributes for the `lost-in-2-ply` task required three weeks. Adapting this set to the slightly different task of learning `lost-in-3-ply` positions took almost two months. Thus for the `lost-in-4-ply` task, which he intended to tackle next, Quinlan experimented with methods for automating the discovery of new useful attributes. However, no results from this endeavor have been published.

A severe problem with this and similar experiments was that although the derived decision trees were shown to be correct and faster in classification than extensive search algorithms, they were also incomprehensible to chess experts. Shapiro and Niblett (1982) tried to alleviate this problem by decomposing it into a hierarchy of smaller sub-problems that could be tackled independently. A set of rules was induced for each of the sub-problems which together yielded a more understandable result. This process of *structured induction* has been employed to learn correct classification procedures for the KPK and the KPa7KR endgames (Shapiro 1987). An endgame expert helped to design the relevant attributes and to structure the search space. The rules for the KPa7KR endgames were generated without using a database as an oracle. The rules were interactively refined by the expert by specifying new examples and suggesting new attributes if the available attributes were not able to discriminate between some of the positionsd. This rule-debugging process was aided by a self-commenting facility that displayed traces of the classification rules in plain English (Shapiro and Michie 1986). A similar semi-autonomous process for refining the attribute set was used by Weill (1994) to generate decision trees for the KQKQ endgame.

However, the problem of decomposing the search space into easily manageable subproblems again is a task that requires extensive collaboration with a human expert. Thus there have been several attempts to automate this process. Paterson (1983) tried to automatically structure the KPK endgame using a clustering algorithm. The results have been negative, as the found hierarchy had no meaning to human experts. Muggleton (1990) has applied DUCE, a machine learning algorithm that is able to autonomously suggest high-level concepts to the user, to the KPa7KR task studied by Shapiro (1987). DUCE looks for common patterns in the rule base (which initially consists of a set of rules each describing one example board position) and tries to reduce the size of the rule base by replacing the found patterns with new concepts. In machine learning the autonomous introduction of new concepts during the learning phase is commonly known as *constructive induction* (Matheus 1989). Using constructive induction DUCE reduces the role of the chess expert to a mere evaluator of the suggested concepts instead of an inventor of new concepts. DUCE structured the KPa7KR task into a hierarchy of 13 concepts defined by a total of 553 rules. Shapiro's solution, however, consisted of 9 concepts with only 225 rules. Nevertheless DUCE's solution was found to be meaningful for a chess expert.

Muggleton (1988) has also proposed a system that uses *sequence induction* to learn strategies that enable it to play a part of the difficult KBBKN endgame. Learning a playing strategy is a harder task than learning a classifier, because the latter does not incorporate the notion of progress. A program that uses a classifier to play a KRK endgame will perform random moves as long as the resulting position is still won, i.e. as long as the opponent is not stalemated or can capture the rook. A program that has access to a winning strategy knows that it has to force the opponent's king towards the edge of board and will proceed to do so. Muggleton's program has induced a set of strategies for freeing a cornered bishop in the KBBKN endgame using an algorithm for inducing regular grammars.

PAL (Morales 1991) is a system for inducing chess patterns in a first-order logic representation. It differs in many aspects from the other approaches described in this section. First of all its

domain is not the induction of endgame classification procedures, but the learning of general patterns like *threat, fork, pin, skewer* etc. from simple example positions. Second it is the only system that is able to use the full power of a first-order logic representation formalism. Thus it is part of the rapidly growing field *inductive logic programming (ILP)* (Muggleton 1992), which is — roughly speaking — concerned with the induction of PROLOG programs. The ability to use background knowledge in the form of PROLOG clauses allows PAL to formulate rules with complex predicates. The rules may even employ a limited look-ahead by including conditions like `make_move(Side,Piece,From,To,Pos,NewPos)` and looking for discriminating patterns in the new position after the specified move has been performed. PAL requires a human tutor to provide a few training examples for the concepts to learn, but it assists the user by generating additional examples with a *perturbation algorithm*. This algorithm takes the current concept description and generates new positions by changing certain aspects like the side to move, the piece to move etc. These examples have to be evaluated by the user before they are used for refining the learned concepts. PAL's architecture is quite similar to Shapiro's learning programs, but it is able to induce concepts in a more powerful description language. In (Morales 1994) this flexibility was demonstrated by using PAL to learn a playing strategy for the KRK endgame. For this task it was trained by a chess expert that provided examples for a set of useful intermediate concepts.

A different problem was tackled by (Bain 1994; Bain and Srinivasan 1995) also using methods from inductive logic programming. An ILP algorithm learned rules for predicting the number of plies to a win by optimal play on both sides in the KRK endgame and found a fairly simple and understandable set of rules that covered the whole example space. We believe that due to its ability to incorporate relational background knowledge inductive logic programming is a very promising line of research for machine learning in chess domains.

4 EXPLANATION-BASED LEARNING

Explanation-based learning (Mitchell et al. 1986) has been devised as a procedure that is able to learn in domains with a rich domain theory. The basic idea is that the domain theory can be used to find an *explanation* for a given example which can then be generalized. This generalized explanation yields a rule with which similar examples can be classified, i.e. examples that share those features that are necessary for the applicability of this particular explanation. Research in explanation-based learning flourished at the end of the eighties. As it seemed to be perfectly applicable to chess (we have a well-defined domain theory, the rules of the game) several authors tried to apply these algorithms to learn useful chunks of knowledge. However, this approach can only be applied to learning simple combinations, which can be explained by looking at the move tree. Interesting patterns, like appropriate pawn moves in certain pawn moves, cannot be explained by the available domain theory and therefore cannot be learned.

A very early explanation-based learning approach to chess has been demonstrated by Jacques Pitrat (Pitrat 1976a; Pitrat 1976b). His program is able to learn definitions for simple concepts like `skewer` or `fork`. The concepts are represented as generalized move trees, so-called *procedures*. For understanding a move the program tries to construct a move tree using already learned procedures. This tree is then simplified (unnecessary moves are removed) and generalized (squares and pieces are replaced with variables according to some predefined rules) in order to yield a new procedure that is applicable to a variety of similar situations. Pitrat's program successfully discovered a variety of useful tactical patterns. However, it turned out that the size of the move trees grow too fast once the program has learned a significant number of patterns of varying degrees of utility. This *utility problem* is a widely acknowledged difficulty for explanation-based learning algorithms (Minton 1990). Interestingly, it has lead Pitrat to the conclusion that the learning module works well, but a more sophisticated method for dealing with the many procedures discovered during learning is needed (Pitrat 1977).

Minton (1984) reports the same problem of learning too many too specialized rules with explanation-based learning in several game domains including chess. Minton's program performs a backward analysis after the opponent has achieved one of its goals (check-mate, winning a piece

etc.). It then identifies the pattern that has triggered the rule which has established that the opponent has achieved a goal. From this pattern all features that have been added through the opponent's last move are removed and all conditions that have been necessary to enable the opponent's last move are added. Thus the program derives a pattern that allows to recognize the danger before the opponent has made the devastating move. If the program's own last move has been forced, the pattern is again changed by deleting the effects of this move and adding the prerequisites for this move. This process of *pattern regression* is continued until one of the program's moves has not been forced. The remaining pattern will then form the condition part of a rule that recognizes the danger before the forced sequence of moves that has achieved the opponent's goal.

However, the limitation to tactical combinations where each of the opponent's moves is forced is too restrictive for the domain of chess. Minton's program could only discover patterns for forced mating sequences. Puget (1987) tried to generalize this work in the following way: All opponent's moves are considered to be optimal, because the opponent has reached one of his goals by this move sequence. When the program tries to regress the pattern of one of its own moves it first picks the move that has actually been played in the game and regresses the pattern over this move. If the move has been forced, this is no different from the method used in (Minton 1984). If, however, the move has not been forced, the program first removes all moves that would not change the regressed pattern. For all remaining moves, i.e. for moves that would refute the following combination, it is checked whether another pattern that has already been learned guarantees the achievement of the same or a better goal in the current position. If so, this pattern is also regressed over the move that lead to it. The union of all regressed patterns forms the new pattern that has to be regressed over the opponent's previous move. Although this approach has been successfully applied to the game of Go-Moku it is doubtful that it would have been equally successful in the more complicated domain of chess.

Puget (1987) states explicitly that the rules derived by his program need not be sufficient for deriving the target concept (as it is usually the case in explanation-based learning). In other words the learned patterns suggest good moves, but do not guarantee the successful outcome of the combination in slightly different situations. Tadepalli (1989) has also acknowledged this problem and proposed a different solution. The philosophy of *Lazy Explanation-Based Learning* is to attack this problem by paying no attention to the possible refutations of a move. The program learns a set of over-general, optimistic plans called *o-plans*. All moves in the plan contribute to the achievement of either the goal itself or one of its preconditions. During actual play a planning module combines all previously learned o-plans that seem to be relevant for the current board situation into so-called *c-plans* which are used for determining the program's next move. Whenever the current c-plan fails because of an unexpected move of the opponent, a new o-plan is learned that incorporates this counter-strategy. Thus, over-general plans are continuously refined by learning new o-plans that will be indexed as counter plans to the original plans. Tadepalli calls this simplifying assumption *Omniscience Assumption*: The planner assumes that it knows all the o-plans necessary to identify the relevant moves for both sides. With an increasing library of o-plans the planner should approximate towards true "omniscience".

Flann (1989) has applied EBL to the acquisition of recognition rules for abstractions. His program, Place is able to recognize many typical, abstract concepts (e.g. `my-king-in-check`) and associate plans and goals with them. The goals are completely instantiated and highly specialized, thus constraining the search during the problem-solving phase. Place does not have to consider all moves in a position, but reasons with abstract operators (e.g. `move-my-king-out-of-check`). Place looks for operators that maintain, destroy or achieve a goal and generalizes the found explanations into operational recognition rules. Complex expressions for the achievement of multiple goals are analyzed and compiled by doing an exhaustive case analysis (Flann 1990). This analysis is able to generate geometrical constraints that can be used as a recognition pattern for the abstract concept.

Flann and Dietterich (1989) demonstrate methods for augmenting EBL with a similarity-based induction module that is able to inductively learn concepts from multiple explanations for several examples. In a first step, the common subtree of the explanations for each of the examples is computed and generalized with EBL. In a second phase this general concept is inductively

specialized by replacing some variables with constants that are common in all examples. While pure EBL is only able to learn the fairly general concept `check-with-bad-exchange` (because it is the only generalization within its search space), IOE (*Induction over Explanations*) is in addition able to learn descriptions for three of its special cases, namely `skewer`, `knight-fork`, and `sliding-fork`.

5 CASE-BASED REASONING

Case-Based Reasoning (CBR) (Kolodner 1992) has been originally developed as a cognitive model of reasoning (Riesbeck and Schank 1989). The basic idea behind CBR is simple: In order to solve a new problem a CBR system tries to remember similar problems that it has previously solved and adapts the solutions of these problems to the new situation. By storing the found solutions in a case base so that they will be available for help in solving future problems the system constantly improves its problem solving skills and thus can be said to "learn". In some way CBR can be viewed as a generalization of "Learning by Analogy", which has already been shortly discussed in (Skiena 1986). However, at that time there was no significant chess application in this area. The author only mentions that the famous PARADISE system (Wilkins 1980) in some way reasons by analogy, but does not incorporate learning.

An early application of case-based reasoning to the domain of chess can be found in (Spohrer 1985). MAPLE (Mistakes As Plan Learning Experiences) learns simple plans for the game of chess. Learning is invoked whenever a disaster occurs, i.e. whenever MAPLE can only choose moves that will result in a loss of material. Then a data compression algorithm brings the crucial moves of the situation into a concise form, using existence and for-all operators. A very simple causal traceback is used to determine the reason for the loss in terms of attacking, skewer and block relations. The result is then abstracted by replacing the location of the pieces with variables and stored as a new plan. The program uses a 2-ply look-ahead for play. When a plan suggests a move, this move is played (move selection). When a plan for the opponent prohibits a move, this move is not made (move rejection). The way MAPLE generalizes plans is explanation-based and shares many similarities with research that we have discussed in section 4, in particular with (Minton 1984).

CASTLE (Krulwich 1993) also relies heavily on previous research in explanation-based learning, but the framework in which CASTLE and MAPLE operate is that of *case-based planning* (Hammond 1989), in which plans that have been generalized from past experiences are used and, whenever they fail, continuously debugged.[1] CASTLE consists of several modules (Krulwich et al. 1995). The threat detection component is a set of condition-action rules, each rule being specialized for recognizing a particular type of threat in the current focus of attention that is determined with another rule-based component. A counterplanning module analyzes the discovered threats and attempts to find countermeasures. CASTLE invokes learning whenever it encounters an *explanation failure*, e.g. when it turns out that the threat detection and the counterplanning components have failed to detect or prevent a threat. In such a case CASTLE uses a self-model to analyze which of its components is responsible for the failure, tries to find an explanation for the failure and then employs explanation-based learning to generalize this explanation (Collins et al. 1993). CASTLE has demonstrated that it can successfully learn plans for interposition, discovered attacks, forks, pins and other short-term tactical threats. From the point of view of learning this approach does not go beyond the capabilities of the explanation-based learning techniques that we have discussed in section 4, but the main focus of the project was to discuss a variety of issues that have to be dealt with in multi-component planning systems.

Kerner (1995) is developing a case-based reasoning program that eventually shall be able to strategically analyze chess positions and suggest possible plans to the user. The system is designed as an educational tool and thus does not need to incorporate search. The basic knowledge consists of a hierarchy of strategic chess concepts (like backward pawns etc.) that has been compiled by an expert. Indexed with each concept is an *Explanation Pattern (XP)* (Schank 1986), roughly speaking a variabilized plan that can be instantiated with the parameters of the current position.

[1] Tadepalli's *Lazy EBL* (see section 4) could also be cast into this framework.

When the system encounters a new position it retrieves XPs that involve the same concept and adapts the plan for the previous case to the new position using generalization, specialization, replacement, insertion and deletion operators. If the resulting XP is approved by a chess expert it will be stored in the case base. Research on this system is still in progress and currently concentrates on the learning of multiple explanation patterns, i.e. the integration of several XPs into the analysis process.

Scherzer et al. (1990) describe an addition to the BEBE chess program that allows it to avoid the repetition of previous mistakes with a technique that uses the program's transposition tables (see also (Slate 1987)). BEBE stores all positions it has encountered during play along with the played move, the search depth, and the move's evaluation. At the beginning of the search the program's transposition table is initialized with the stored positions. Whenever one of them is encountered during tree search the program will utilize the stored evaluation which results from a deeper search. Thus the program will be able to search deeper whenever it is at or near positions that have occurred in prior games. It has been experimentally confirmed that BEBE's learning in fact improves its score considerably when playing 100-200 games against the same opponent.

Flinter and Keane (1995) present TAL, a recent program that uses a case-based approach for reducing the number of moves that are considered during tree search. TAL has access to a library of 4,533 base chunks (i.e. meaningful groups of pieces) that it has generated using 350 chess games of ex-world-champion Mikhail Tal. These chunks will subsequently be used to construct a case base of chess positions that can be used by a playing module to retrieve positions with a similar chunk structure and limit the number of considered moves in a new position to adaptations of the moves that have been played in the positions that correspond to the retrieved cases. However, the playing module has not yet been implemented which makes it hard to evaluate the feasibility of this approach. TAL's architecture was motivated by psychological research on human chess playing (for an overview see (Holding 1985)) and the implementation of the chunking algorithm seems to produce plausible results. Technically the approach is quite similar to (Schaeffer 1988), where positions in the game graph are represented with a feature vector that encodes important aspects of each position, like material balance, pawn structure etc. When a new position is analyzed, the program computes its feature vector and retrieves all positions with identical or almost identical feature vectors from the game tree. This ensures that the most important features of the retrieved positions match the current position. The moves played in the most similar (80% of the pieces have to be on identical squares) of the retrieved positions will get a bonus of a quarter pawn during regular tree search. Both approaches learn from the experience of other players by using a database of games as an aid in the move selection process.

CHUMP (Gobet and Jansen 1994) is another very interesting approach to build a playing chess program based on psychological results on human chess playing. CHUMP uses an eye-movement simulator (Simon and Barenfeld 1969) to scan the board into a fixed number (20) of meaningful chunks. New chunks will be added into a discrimination net (Simon and Gilmartin 1973) based on the EPAM model of memory and perception (Feigenbaum 1961). During the learning phase the move that has been played in the position is added into a different discrimination net and is linked to the chunk that contains the moved piece at its original location. In the playing phase the retrieved patterns are checked for associated moves (note that only about 10% of the chunks have associated moves). The move that is suggested by the most chunks will be played. In one experiment CHUMP was trained on 300 games of ex-world champion Mikhail Tal. Tests on seen and unseen positions show continuous improvement during learning. After learning from all positions from 300 games in 28.7% of unseen positions the correct move is in the set of retrieved moves. However, in only 4 of the 143 test positions from two complete games CHUMP's favorite move has actually been played. In a test on the 24 Bratko-Kopec positions (Kopec and Bratko 1982) CHUMP achieved a rating of below 1500, but its performance on the lever positions (where positional chunks can be very helpful) was much better than its performance on the tactical positions (where it is severely handicapped because it doesn't use any search). In a second experiment CHUMP was trained on several games in the KQKR endgame. The moves played in master games were almost always among the moves retrieved by CHUMP, when tested on the same positions that it has learned from. This percentage decreases considerably when testing on unseen games. One of the major

deficiencies of CHUMP seems to be that the number of learned chunks increases almost linearly with the number of seen positions and that the number of chunks that have an associated move remains constant over time. In our opinion this suggests that CHUMP always learns new patterns, i.e. that its capabilities for reusing already learned patterns are limited. However, CHUMP is the first chess program based on a model of human chess memory and perception that is able to play a complete game.

A similar approach to a chess playing program based on the use of psychologically motivated chunks is the Inductive Adversary Modeler (IAM) (Walczak 1991; Walczak and Dankel 1991). IAM's goal is to acquire a set of patterns that reflect the opponent's typical play. Thus it can be regarded as part of the growing research in *opponent modeling* (see e.g. (Iida et al. 1995)). When it is IAM's turn to move it looks for nearly completed chunks, i.e. chunks that can be completed by one of the opponent's moves, and predicts that the opponent will play this move. Multiple move predictions are resolved heuristically with preference being given to bigger and more reliable chunks. Using this method IAM was able to correctly predict more than 10% of the moves in games of several grandmasters including Botwinnik, Karpov, and Kasparov.[2]

6 EVALUATION FUNCTION LEARNING

The tuning of evaluation functions of conventional chess programs has become one of the most promising directions for the application of machine learning methods in computer chess. Since the days of the famous learning checker playing program by Samuel (1959) the idea of using games from human experts, against human opponents, or collected by self-play to optimize the feature weights of an evaluation function has been present in computer game playing. Nitsche (1982) has suggested a way for minimizing the least squares error of the evaluation function over a collection of games. The proposed algorithm was to choose the weights for the evaluation function in such a way that the sum over all moves of the squared differences of the move's heuristic value and a *selection function* is minimized. The selection function is defined to be 1 if this move has been selected by the opponent and 0 otherwise. The feasibility of this approach has been demonstrated with a simple endgame example.

Marsland (1985) extends this work by suggesting that the ordering of the moves by the evaluation function should be included into the learning process as well. He proposes cost functions that give a higher penalty if the chosen move has a low ranking than if it is considered as one of the best moves. He also favorizes a cost function that makes sure that changing the rank of a move near the top has a bigger effect than changing the rank of a move near the bottom.

Similarly, van der Meulen (1989) criticizes Nitsche's approach because of the use of a discrete selection function. Instead he proposes to calculate the weights for the evaluation functions using a set of inequalities that define the region in parameter space in which the evaluation function would choose the same move as a grandmaster did in a certain position. These regions are simplified to hyperrectangles. The intersection of several such hyperrectangles defines a region where the evaluation function finds the correct move for several test positions. Van der Meulen's algorithm greedily identifies several such regions and constructs a suitable evaluation function for each of them. New positions are then evaluated by firstly identifying an appropriate region and then using the associated evaluation function. However, this algorithm has not been tested, and it is an open question whether the number of optimal regions can be kept low enough for an efficient application in practice.

Hsu et al. (1990a) used an automatic tuning approach for their world champion program DEEP THOUGHT, because they were convinced that it is infeasible to correctly reflect the interactions between the more than 100 weights of the evaluation functions. Like Nitsche (1982) they chose to compute the weights by minimizing the squared error between the program's and a grandmaster's choice of moves. They implemented an efficient algorithm that compares the *dominant* position (i.e. the leaf position that was responsible for the evaluation of the root) of the subtree starting

[2]IAM does not always predict a move; thus the percentage of correctly predicted moves when it made a prediction is even higher.

with the grandmaster's move to the dominant position of any alternative move at a shallow 5 to 6 ply search. If an alternative move's dominant position gets a higher evaluation an appropriate adjustment direction is computed in the parameter space and the weight vector is adjusted a little into that direction. The authors had also experimented with a simple hill-climbing algorithm using an approximation of the true value of the evaluation function that was obtained by a deeper search. Although this approach turned out to be too inefficient to be used for tuning the weights, it proved to be useful in evaluating the implemented tuning algorithm. The authors were convinced that their automatically tuned evaluation function is no worse than the hand-tuned functions of their academic competitors. However, they also remarked that the evaluation functions of top commercial chess programs are still beyond them, but hoped to close the gap soon (Hsu et al. 1990b).

Tunstall-Pedoe (1991) has tried to optimize the weights of an evaluation function with *genetic algorithms*. A genetic algorithm (Goldberg 1989) is a robust optimization method that avoids to get stuck in local optima. The algorithm maintains a set of weight vectors called a *generation*. The next generation is computed by randomly changing some weights (*mutation*) of a member of the last generation or by exchanging some weights between two members (*cross-over*). The probability that members of a generation are selected for these operations (and that they will survive in the next generation) is proportional to their *fitness*. The fitness of a certain parameter set was estimated with the percentage of test positions for which the evaluation function chose the same move[3] as the grandmaster at a random sample of 500 positions chosen from 389 grandmaster games. The program acquired a set of weights that performed no worse than the values that have been manually set by the author of the used chess program, but surprisingly showed little correlation with the author's estimates.

The price that has to be paid for the avoidance of local optima with the use of a genetic algorithm is its inefficiency. (van Tiggelen and van den Herik 1991) have concluded from a study in the KNNKP(h) endgame that genetic learning is too inefficient to scale up to a middle-game application with a reasonable number of test positions and a reasonable search depth for move evaluation. (van Tiggelen 1991) therefore looked for a more efficient approach and tried to optimize evaluation functions with a *neural network*. The network was trained to predict the performance of the weights on a set of test positions as in (Tunstall-Pedoe 1991), but the positions were evaluated under tournament conditions (i.e. $\approx$ 3 minutes per position) as in (van Tiggelen and van den Herik 1991). From his description it is not entirely clear how the author used this network, but we presume he used the weights in the evaluation function for which the network predicted a high performance. His conclusions were that his neural-net optimization method is not only more efficient than the use of genetic algorithm, but also yields much better results in the KNNKP(h) test domain. Schmidt (1993) has performed a variety of experiments, in which he tried neural networks to learn an evaluation function for middle-game chess positions, but with rather disappointing results. His conclusion was to switch to temporal difference learning (Schmidt 1994).

The formulation of the *temporal difference learning* paradigm (Sutton 1988) was a major advance for machine learning in strategic game playing. Although the basic idea of TD learning has already been employed in Samuel's famous checkers player (Samuel 1959), it has been forgotten in computer game playing research until the success of Tesauro's master-level backgammon program TD-GAMMON (Tesauro 1992; Tesauro 1994). This achievement naturally lead to attempts to apply temporal difference learning to computer chess.

TD learning attempts to solve the hard problem of obtaining correct evaluations for middle-game positions that can be used for training a supervised learning algorithm. As in chess the outcome of the game can only be determined at its end, it is hard to identify the moves that have contributed the most to this outcome. This is also known as the *credit assignment problem*. The proposed solution is that instead of learning from a set of preselected and preevaluated training positions, the algorithm tries to minimize the difference between successive position evaluations. For example if the program constantly thinks it has an equal position and suddenly discovers that

[3] At an efficient 1 ply + quiescence search.

it will loose a piece, something must have gone wrong with the evaluations of the previous positions and the weights should be adjusted to reflect this fact. The TD(λ) learning framework takes this into account by adjusting the weights of the evaluation function by an amount that depends on the difference between the most recent position evaluation and the preceding estimates. In order to decrease the influence of positions that have occurred longer ago, the influence of each position is weighted by λ^n where n is the number of moves that have been played since this position and λ is a user-settable parameter ($0 \leq \lambda \leq 1$). If λ is set to 1, all positions in the game are weighted equally, while with $\lambda = 0$ (*Q-learning*) only the current position evaluation is used for adjustment. In that case the evaluation function is trained with its own value one move later in the game.

Gherrity (1993) has integrated Q-learning with *consistency search* (Beal 1980) in the SAL ("Search and Learning") system. A neural network evaluation function[4] is trained with self play, where the evaluation function of the next board position is used as the target value for the evaluation value of the current position. SAL played several thousand games against GNUCHESS and was eventually able to achieve 8 draws by perpetual check. A more detailed evaluation nevertheless showed that SAL was able to learn, as the average length of the game and the average number of captured pieces increased.

A far better performance was achieved by NEUROCHESS (Thrun 1995), which uses an interesting approach to combine explanation-based and neural network learning in a temporal difference learning framework. Domain-specific chess knowledge is represented with the help of a neural network version of explanation-based learning (Thrun and Mitchell 1993). This network is trained to predict the board position two plies after the current position using 120,000 expert games. A separate network is used for learning the evaluation function which is trained with its own estimates for the predicted positions using a TD(0) algorithm. NEUROCHESS learned from playing against GNUCHESS. It scored only 6 points in the first 200 games (3%), but steadily increased this ratio. After 2000 games is was able to score 25% in the last 400 games.

MORPH (Levinson and Snyder 1991; Gould and Levinson 1994) is arguably the most advanced (not necessarily the strongest) temporal difference learning system in the chess domain. Its major achievement is that instead of only optimizing the weights of a set of predefined positional features it also automatically constructs the features during learning. The basic representational entity of MORPH are *pattern-weight pairs (pws)*. Patterns are represented as conceptual position graphs, the nodes being pieces and important squares and the edges being attack relationships. These patterns are created by choosing a random size n and adding the best n nodes to the graph according to some predefined ordering heuristic. Patterns can subsequently be generalized by extracting the common subtree of two patterns with similar weights or specialized by adding nodes and edges to the graph whenever a weight must be updated by a large amount, which indicates inaccuracy. Different patterns can be tied together by *reverse engineering*, so that they result in a chain of actions. This process is quite similar to explanation-based learning. Genetic cross-over and mutation operators are also planned to be included in the near future. Pattern weights are modified by a general TD(λ) algorithm, where the final result of the game is propagated down to all patterns that occurred in the positions of the game. A major difference to common approaches is that each pattern has its own learning rate which is adjusted with a *simulated annealing* scheme (i.e. the more frequently a pattern is updated, the slower becomes its learning rate). MORPH evaluates a move by combining the weights of all matched patterns into a single evaluation function value and selecting the move with the best value, i.e. it performs only a 1-ply look-ahead. The system has been tested against GNUCHESS and was able to beat it occasionally. The special patterns that MORPH has learned for the estimation of differences in material proved to be consistent with common chess knowledge. MORPH soon started to play reasonable opening moves, although no information about development or center control has been added to the system. However, a major problem of MORPH is that although it is able to delete useless patterns, it will still be swamped by too many patterns, a problem that is common to all pattern-learning systems.

[4] Actually SAL uses two evaluation functions, one for each side.

7 CONCLUSION

Let us now reevaluate Skiena's conclusions of 1986. We have seen that several competitive programs have experimented with learning techniques. DEEPTHOUGHT's evaluation function has been automatically tuned on a database of grandmaster games and BEBE used a learning algorithm to avoid the repetition of mistakes in a match. Although the use of machine learning methods still is not common in competitive computer chess programs, the situation has improved in the last decade.

However, we hope to have convinced the reader that Skiena's second conclusion can be rejected in the light of the research of the last decade. Due to their ability to incorporate relations into the learning process, inductive logic programming algorithms are particularly suitable for learning in game playing domains. The application of temporal-difference learning to computer chess is still in its infancy, but its success in Tesauro's backgammon program gives reason for optimism. We also have seen that research in pattern-based computer chess programs has recently gained interest. In particular in educational tools the use of patterns and plans will be crucial as will the use of machine learning techniques to acquire these patterns. A major problem that has to be addressed here is the abundance of possible patterns that has to be met with clever evaluation techniques. Research in psychologically motivated chess programs has also regained popularity after it has stagnated in the early 70's. We consider research in this direction as a particularly promising endeavor which eventually may be able to shed light on some aspects of human cognition as well as produce chess programs that overcome some of the strategical weaknesses of brute-force approaches.

Acknowledgements

An overview of this sort can never be complete. I have tried to include every significant research project in this area which I am aware of and which I had access to. I have put a more complete list of references in the area of *Machine Learning in Strategic Game Playing* on the WWW at `http://www.ai.univie.ac.at/~juffi/lig/lig.html` (with links to many on-line papers, a search facility, and a `bib`-file). I would like to thank all people who have contributed to this collection, in particular Jay Scott who maintains an excellent WWW-site for this research field at `http://forum.swarthmore.edu/~jay/learn-game/`. I would also like to thank Bernhard Pfahringer for comments on an earlier version of this paper. This research is sponsored by the Austrian *Fonds zur Förderung der Wissenschaftlichen Forschung (FWF)*. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry of Science and Research.

References

Bain, M. and A. Srinivasan (1995). Inductive logic programming with large-scale unstructured data. In D. Michie, K. Furukawa, and S. Muggleton (Eds.), *Machine Intelligence 14*. Oxford University Press.

Bain, M. E. (1994). *Learning Logical Exceptions in Chess*. Ph. D. thesis, Department of Statistics and Modelling Science, University of Strathclyde, Scotland.

Beal, D. F. (1980). *Advances in Computer Chess 2*, Chapter An Analysis of Minimax, pp. 103–109. Edinburgh University Press. Editor: M. R. B. Clarke.

Collins, G., L. Birnbaum, B. Krulwich, and M. Freed (1993). The role of self-models in learning to plan. In *Foundations of Knowledge Acquisition: Machine Learning*, pp. 83–116. Kluwer.

deGroot, A. D. (1965). *Thought and Choice in Chess*. The Hague: Mouton.

Feigenbaum, E. (1961). The simulation of verbal learning behavior. In *Proceedings of the Western Joint Computer Conference*, pp. 121–132. Reprinted in J.W. Shavlik & T.G. Dietterich (eds.), Readings in Machine Learning, Morgan Kaufmann 1990.

Flann, N. S. (1989). Learning appropriate abstractions for planning in formation problems. In A. M. Segre (Ed.), *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 235–239. Morgan Kaufmann.

Flann, N. S. (1990). Applying abstraction and simplification to learn in intractable domains. In B. W. Porter and R. Mooney (Eds.), *Proceedings of the 7th International Conference on Machine Learning*, pp. 277–285. Morgan Kaufmann.

Flann, N. S. and T. G. Dietterich (1989). A study of explanation-based methods for inductive learning. *Machine Learning 4*, 187–226.

Flinter, S. and M. T. Keane (1995). Using chunking for the automatic generation of cases in chess. In M. Veloso and A. Aamodt (Eds.), *Proceedings of the 1st International Confernece on Case Based Reasoning (ICCBR-95)*. Springer Verlag.

George, M. and J. Schaeffer (1990). Chunking for experience. *ICCA Journal 13*(3), 123–132.

Gherrity, M. (1993). *A Game-Learning Machine*. Ph. D. thesis, University of California, San Diego, CA.

Gobet, F. and P. Jansen (1994). Towards a chess program based on a model of human memory. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 35–60. University of Limburg.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.

Gould, J. and R. Levinson (1994). Experience-based adaptive search. In R. Michalski and G. Tecuci (Eds.), *Machine Learning: A Multi-Strategy Approach*, pp. 579–604. Morgan Kaufmann.

Hammond, K. J. (1989). *Case-Based Planning: Viewing Planning as a Memory Task*. Boston, MA: Academic Press.

Holding, D. H. (1985). *The Psychology of Chess Skill*. Lawrence Erlbaum Associates.

Hsu, F., T. S. Anantharaman, M. S. Campbell, and A. Nowatzyk (1990a). Deep thought. In T. A. Marsland and J. Schaeffer (Eds.), *Computers, Chess, and Cognition*, Chapter 5, pp. 55–78. Springer-Verlag.

Hsu, F., T. S. Anantharaman, M. S. Campbell, and A. Nowatzyk (1990b, October). A grandmaster chess machine. *Scientific American 263*(4), 44–50.

Iida, H., J. Uiterwijk, H. van den Herik, and I. Herschberg (1995). Thoughts on the application of opponent-model search. In H. van den Herik and J. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 61–78. Maastricht, Holland: Rijksuniversiteit Limburg.

Kerner, Y. (1995). Learning strategies for explanation patterns: Basic game patterns with application to chess. In M. Keane, J. Haton, and M. Manago (Eds.), *Proceedings of the 1st International Conference on Case-Based Reasoning (ICCBR-95)*, Lecture Notes in Artificial Intelligence, Berlin. Springer-Verlag.

Kolodner, J. L. (1992). *Case-Based Reasoning*. Morgan Kaufmann.

Kopec, D. and I. Bratko (1982). The Bratko-Kopec experiment: A comparison of human and computer performance in chess. In M. Clarke (Ed.), *Advances in Computer Chess 3*, pp. 57–72. Oxford: Pergamon.

Krulwich, B., L. Birnbaum, and G. Collins (1995). Determining what to learn through component-task modeling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pp. 439–445.

Krulwich, B. T. (1993). *Flexible Learning in a Multi-Component Planning System*. Ph. D. thesis, The Institute for the Learning Sciences, Northwestern University, Evanston, IL. Technical Report #46.

Levinson, R. A. and R. Snyder (1991). Adaptive pattern-oriented chess. In L. Birnbaum and G. Collins (Eds.), *Proceedings of the 8th International Workshop on Machine Learning*, pp. 85–89. Morgan Kaufmann.

Marsland, T. A. (1985). Evaluation-function factors. *ICCA Journal 8*(2), 47–57.

Matheus, C. J. (1989). A constructive induction framework. In *Proceedings of the 6th International Workshop on Machine Learning*, pp. 474–475.

Michalski, R. and P. Negri (1977). An experiment on inductive learning in chess endgames. In Elcock and D. Michie (Eds.), *Machine Intelligence 8*, pp. 175–192. Edinburgh University Press.

Michie, D. and I. Bratko (1991, March). Comments to 'chunking for experience'. *ICCA Journal 18* (1), 18.

Minton, S. (1984). Constraint based generalization: Learning game playing plans from single examples. In *Proceedings of the 2nd National Conference on Artificial Intelligence*, Austin, TX, pp. 251–254.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence 42*, 363–392.

Mitchell, T. M., R. M. Keller, and S. Kedar-Cabelli (1986). Explanation-based generalization: A unifying view. *Machine Learning 1*(1), 47–80.

Morales, E. (1991). Learning features by experimentation in chess. In *Proceedings of the 5th European Working Session on Learning*, pp. 494–511. Springer Verlag.

Morales, E. (1994). Learning patterns for playing strategies. *ICCA Journal 17*(1), 15–26.

Muggleton, S. (1988). Inductive acquisition of chess strategies. In J. E. Hayes, D. Michie, and J. Richards (Eds.), *Machine Intelligence 11*, Chapter 17, pp. 375–387. Clarendon Press.

Muggleton, S. (1990). *Inductive Acquisition of Expert Knowledge*. Turing Institute Press. Addison-Wesley.

Muggleton, S. (Ed.) (1992). *Inductive Logic Programming*. London: Academic Press Ltd.

Nitsche, T. (1982). A learning chess program. In M. R. B. Clarke (Ed.), *Advances in Computer Chess 3*, pp. 113–120. Pergamon Press.

Nunn, J. (1992). *Secrets of Rook Endings*. Batsford.

Nunn, J. (1994a). Extracting information from endgame databases. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 19–34. University of Limburg.

Nunn, J. (1994b). *Secrets of Pawnless Endings*. Batsford.

Paterson, A. (1983). An attempt to use CLUSTER to synthesise humanly intelligible subproblems for the KPK chess endgame. Technical Report UIUCDCS-R-83-1156, University of Illinois, Urbana, IL.

Pitrat, J. (1976a). A program to learn to play chess. In *Pattern Recognition and AI*, pp. 399–419. Academic Press.

Pitrat, J. (1976b). Realization of a program learning to find combinations at chess. In J. Simon (Ed.), *Computer Oriented Learning Processes*. Noordhoff.

Pitrat, J. (1977). A chess combination program which uses plans. *Artificial Intelligence 8*, 275–321.

Puget, J.-F. (1987). Goal regression with opponent. In *Progress in Machine Learning*, pp. 121–137. Sigma Press.

Quinlan, J. R. (1983). Learning efficient classification procedures. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, pp. 463–482. Palo Alto: Tioga.

Riesbeck, C. K. and R. C. Schank (1989). *Inside Case-Based Reasoning*. Hillsdale, NJ: Lawrence Erlbaum Associates.

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

Schaeffer, J. (1988). Learning from (other's) experience. In *Proceedings of the AAAI Spring Symposium on Computer Game Playing*, pp. 51–53.

Schank, R. C. (1986). *Explanation Patterns: Understanding Mechanically and Creatively*. Hillsdale, NJ: Lawrence Erlbaum.

Scherzer, T., L. Scherzer, and D. Tjaden (1990). Learning in BEBE. In T. A. Marsland and J. Schaeffer (Eds.), *Computers, Chess, and Cognition*, Chapter 12, pp. 197–216. Springer Verlag.

Schmidt, M. (1993). Neural networks and chess. Master's thesis, Computer Science Department, University of Aarhus, Aarhus, Denmark.

Schmidt, M. (1994). Temporal-difference learning and chess. Technical report, Computer Science Department, University of Aarhus, Aarhus, Denmark.

Shapiro, A. D. (1987). *Structured Induction in Expert Systems*. Turing Institute Press. Addison-Wesley.

Shapiro, A. D. and D. Michie (1986). A self commenting facility for inductively synthesized endgame expertise. In D. F. Beal (Ed.), *Advances in Computer Chess 4*, pp. 147–165. Oxford: Pergamon.

Shapiro, A. D. and T. Niblett (1982). Automatic induction of classification rules for a chess endgame. In M. R. B. Clarke (Ed.), *Advances in Computer Chess 3*, pp. 73–92. Oxford: Pergamon.

Simon, H. A. and M. Barenfeld (1969). Information-processing analysis of perceptual processes in problem solving. *Psychological Review 76*(5), 473–483.

Simon, H. A. and K. Gilmartin (1973). A simulation of memory for chess positions. *Cognitive Psychology 5*, 29–46.

Skiena, S. S. (1986). An overview of machine learning in computer chess. *ICCA Journal 9*(1), 20–28.

Slate, D. J. (1987). A chess program that uses its transposition table to learn from experience. *ICCA Journal 10*(2), 59–71.

Spohrer, J. C. (1985). Learning plans through experience: A first pass in the chess domain. In D. P. Casasent (Ed.), *Intelligent Robots and Computer Vision*, Volume 579 of *Proceedings of the SPIE – The International Society for Optical Engineering*, pp. 518–527.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning 3*, 9–44.

Tadepalli, P. (1989). Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the 11th International Joint Conference on AI*, pp. 694–700. Morgan Kaufmann.

Tesauro, G. (1992). Temporal difference learning of backgammon strategy. *Proceedings of the 9th International Conference on Machine Learning 8*, 451–457.

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation 6*(2), 215–219.

Thrun, S. (1995). Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen (Eds.), *Advances in Neural Information Processing Systems 7*.

Thrun, S. B. and T. M. Mitchell (1993). Integrating inductive neural network learning and explanation-based learning. In R. Bajcsy (Ed.), *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp. 930–936. Morgan Kaufmann.

Tunstall-Pedoe, W. (1991). Genetic algorithms optimizing evaluation functions. *ICCA Journal 14*(3), 119–128.

van der Meulen, M. (1989). Weight assessment in evaluation functions. In D. F. Beal (Ed.), *Advances in Computer Chess 5*, pp. 81–89. Amsterdam: Elsevier.

van Tiggelen, A. (1991). Neural networks as a guide to optimization. the chess middle game explored. *ICCA Journal 14*(3), 115–118.

van Tiggelen, A. and H. J. van den Herik (1991). ALEXS: An optimization approach for the endgame KNNKP(h). In D. F. Beal (Ed.), *Advances in Computer Chess 6*, pp. 161–177. Chichester: Ellis Horwood.

Walczak, S. (1991). Predicting actions from induction on past performance. In L. Birnbaum and G. Collins (Eds.), *Proceedings of the 8th International Workshop on Machine Learning*, pp. 275–279. Morgan Kaufmann.

Walczak, S. and D. D. Dankel (1991). Acquiring tactical and strategic knowledge with a generalized method for chunking of game pieces. *International Journal of Intelligent Systems*.

Weill, J.-C. (1994). How hard is the correct coding of an easy endgame. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk (Eds.), *Advances in Computer Chess 7*, pp. 163–176. University of Limburg.

Wilkins, D. E. (1980). Using patterns and plans in chess. *Artificial Intelligence 14*(3), 165–203.

Zobrist, A. L. and F. R. Carlson (1973, June). An advice-taking chess computer. *Scientific American*, 93–105.