(Tesauro, 1992) Gerald Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 1992. To appear in Special Issue on reinforcement learning, Richard Sutton, editor.

(Thompson and Roycroft, 1983) K. Thompson and A. J. Roycroft. A prophesy fulfilled. *EndGame*, 5(74):217–220, 1983.

(Wilkins, 1980) D. Wilkins. Using patterns and plans in chess. *Artificial Intelligence*, 14(2):165–203, 1980.

(Yee *et al.*, 1990) R. C. Yee, Sharad Saxena, Paul E. Utgoff, and Andrew G. Barto. Explaining temporal differences to create useful concepts for evaluating states. In *Proceedings of the Eighth National Conference on AI*, Menlo Park, 1990. American Association for Artificial Intelligence, AAAI Press/The MIT Press.

(Zobrist and Carlson, 1973) A. L. Zobrist and D. R. Carlson. An advice-taking chess computer. *Scientific American*, 228:92–105, June 1973.

(Metropolis *et al.*, 1953) N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1091, 1953.

(Michalski, 1983) R. S. Michalski. A theory and methodology of inductive learning. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine learning: An Artificial Intelligence Approach.* Tioga Press, 1983.

(Michie and Bratko, 1987) D. Michie and I. Bratko. Ideas on knowledge synthesis stemming from the KBBKN endgame. *Intern. Computer Chess Assoc. Journal*, 10(1):3–13, 1987.

(Minton, 1984) S. Minton. Constraint based generalization- learning game playing plans from single examples. In *Proceedings of AAAI-84*, pages 251–254. AAAI, 1984.

(Mitchell *et al.*, 1986a) T. M. Mitchell, J. G. Carbonell, and R. S. Michalski, editors. *Machine Learning: A Guide to Current Research.* Kluwer Academic Publishers, 1986.

(Mitchell *et al.*, 1986b) T. M. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation based generalization: A unifying view. In *Machine Learning 1*, pages 47–80. Kluwer Academic Publishers, Boston, 1986.

(Muggleton, 1988) S. H. Muggleton. Inductive acquisition of chess strategies. In D. Michie J. E. Hayes and J. Richards, editors, *Machine Intelligence 11*, pages 375–389. Oxford University Press, Oxford, 1988.

(Niblett and Shapiro, 1981) T Niblett and A. Shapiro. Automatic induction of classification rules for chess endgames. Technical Report MIP-R-129, Machine Intelligence Research Unit, University of Edinburgh, 1981.

(Pitrat, 1976) J. Pitrat. A program for learning to play chess. In *Pattern Recognition and Artificial Intelligence.* Academic Press, 1976.

(Quinlan, 1986) J. R. Quinlan. Induction on decision trees. *Machine Learning*, 1:81–106, 1986.

(Rendell and Seshu, 1990) L. Rendell and R. Seshu. Learning hard concepts through constructive induction: Framework and rationale. *Computational Intelligence*, 6(4):247–270, 1990.

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

(Samuel, 1967) 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.

(Shannon, 1950) C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(7):256–275, 1950.

(Sutton, 1988) R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988.

(Tadepalli, 1989) P. Tadepalli. Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989. Morgan Kaufmann.

(Tesauro and Sejnowski, 1989) G. Tesauro and T. J. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39:357–390, 1989.

References

(Birnbaum and Collins, 1991) A.L. Birnbaum and G.C. Collins, editors. *Proceedings of the Eighth International Workshop on Machine Learning.* Morgan Kaufmann, San Mateo, CA., 1991. See part 3, pages 117-232.

(Botvinnik, 1984) M. Botvinnik. *Computers in Chess.* Springer-Verlag, 1984.

(Christensen and Korf, 1986) J. Christensen and R. Korf. A unified theory of heuristic evaluation. In *AAAI-86*, 1986.

(Davis and Steenstrup, 1987) Lawrence Davis and Martha Steenstrup. Genetic algorithms and simulated annealing: An overview. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, Research Notes in Artificial Intelligence. Morgan Kaufmann Publishers, 1987.

(Epstein, 1990) S. L. Epstein. Learning plans for competitive domains. In *Proceedings of the Seventh International Conference on Machine Learning*, June 1990.

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

(Fox and James, 1987) M. Fox and R. James. *The Complete Chess Addict.* Faber and Faber, London, 1987.

(Glover, 1987) David E. Glover. *Genetic Algorithms and Simulated Annealing*, chapter Chapter 1, Solving a Complex Keyboaard Configuation Problem Through Generalized Adaptive Search. Research Notes in Artificial Intelligence. Morgan Kaufmann Publishers, 1987.

(Holland, 1975) J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.

(Kirkpatrick *et al.*, 1983) S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

(Lee and Mahajan, 1988) K. F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36:1–25, 1988.

(Levinson and Snyder, 1991) R. Levinson and R. Snyder. Adaptive pattern oriented chess. In *Proceedings of AAAI-91*, pages 601–605. Morgan-Kaufman, 1991.

(Levinson *et al.*, 1992) R. Levinson, B. Beach, R. Snyder, T. Dayan, and K. Sohn. Adaptive-predictive game-playing programs. *Journal of Experimental and Theoretical AI*, 1992. To appear. Also appears as Tech Report UCSC-CRL-90-12, University of California, Santa Cruz.

(Levinson, 1989a) Robert Levinson. Pattern formation, associative recall and search: A proposal. Technical report, University of California at Santa Cru, 1989.

(Levinson, 1989b) Robert Levinson. A pattern-weight formulation of search knowledge. Technical Report UCSC-CRL-91-15, University of California Santa Cruz, 1989. Revision to appear in Computational Intelligence.

(Levinson, 1991) R. Levinson. A self-organizing pattern retrieval system and its applications. *Internation Journal of Intelligent Systems*, 6:717–738, 1991.

Acknowledgments

Thank you to Jeff Keller for constructing the new evaluation function, to Paul Zola and Kamal Mostafa for the initial Morph implementation, and to Richard Sutton for sharing our enthusiasm for reinforcement learning. Finally, we would like to thank Richard Snyder for valuable editing assistance. The research was partially funded by NSF grant IRI-9112862 and a Faculty Research Grant from the University of California.

- Guided by appropriate performance measures, modification and testing of the system proceeds systematically.
- Interesting ideas arise directly as a result of taking the multi-strategy view. The goal is to exploit the strength of individual methods while eliminating their weaknesses. Some examples:

    1. *The genetic mutation operator described in Chapter 0.3.3.*
    2. *Higher level concepts via hidden units.* Once a good set of patterns has been obtained it may be possible for the system to develop a more sophisticated evaluation function. This function, patterned after neural nets, would have hidden units that correspond to higher level interactions between the patterns. For example, conjunctions and disjunctions may be realized and given weights different from that implied by their components.
    3. *Clarity of system's knowledge* The "meaning" of hidden units to which weights are associated in neural nets is usually not clear, whereas in experience-based systems it is specific structures that are given weights. Indeed, it is the transparency of Morph's knowledge that has allowed its learning mechanisms to be fine tuned; with various system utilities it is possible to ascertain exactly why Morph is selecting one move over another.

6. Conclusions and Ongoing Directions

The development of a computational framework for experience-based learning is a difficult but important challenge. Here, we have argued for the necessity of a multi-strategy approach: At the least, an adaptive search system requires mechanisms for credit assignment, feature creation and deletion, weight maintenance and state evaluation. Further, it has been demonstrated that the TD error measure can provide a mechanism by which the system can monitor its own error rate and steer itself to smooth convergence. The error rate provides a metric more refined but well-correlated with the reinforcement values and more domain-specific metrics. Finally, in a system with many components (pws) to be adjusted, learning rates should be allowed to differ across these components. Simulated annealing provides this capability.

APS has produced encouraging results in a variety of domains studied as classroom projects (Levinson *et al.*, 1992), including Othello, Tetris, 8-puzzle, Tic-Tac-Toe, Pente, image alignment, Missionary and Cannibals and more. Currently, others are studying the application of the Morph-APS shell[1] to GO, Shogi and Chinese Chess. Here we have used the Morph project to illustrate the potential for these methods. But clearly, much more distance remains to be covered before Morph or other experience-based systems will learn from experience with nearly the efficiency that humans do. To achieve this, substantial refinement of the learning mechanisms and an enhancement of their mutual cooperation is required. Undoubtedly, advances in pattern-based knowledge representation and associative memory will also be required. This model of experience-based learning has pursued a largely syntactic approach to codifying and exploiting experience. What role can and should semantic knowledge serve?

A limitation of the problem-solving system presented here is the reliance on full-width 1-ply search. This is wasteful: many moves considered may be irrelevant whereas other moves may require further search to determine their suitability. That is, at times the use of search may be more economical than developing patterns to make fine distinctions. A selective search more akin to human analysis is desirable for more effective processing. Can such selectivity also be learned from experience? This avenue is currently being pursued.

The key to experience-based learning, beyond recent work in constructive induction (Birnbaum and Collins, 1991; Michalski, 1983; Rendell and Seshu, 1990), is that the system is given responsibility for both the structure and significance of learned knowledge. Experience-based learning should have application far beyond chess. Consider a robot (e.g. a Mars rover) that must learn to survive and manage effectively in a new environment, or a machine tutor that must learn which forms of instruction and examples work better than others. Finally, it may be possible to get organic synthesis systems to improve search time with experience using graph methods similar to Morph (Levinson, 1991).

The following points are worth remembering:

- In combining the many learning methods for experience-based learning the methods are not taken as they are normally used but their essence is extracted and combined beneficially.

[1]Now publicly available via anonymous ftp from ftp.cse.ucsc.edu. The file, morph.tar.Z, is in directory /pub/morph/.

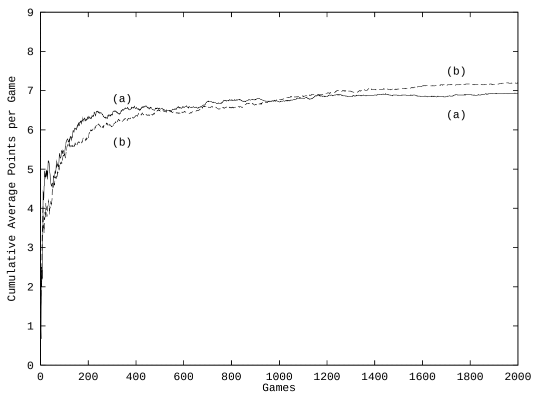

Figure 5.1: Cumulative average of two versions of Morph.

Version (a) is a basic Morph. Version (b) has the reverse node ordering pattern addition scheme added.

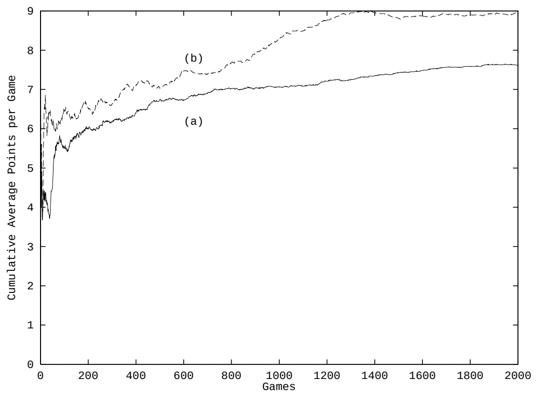

Figure 5.2: Cumulative average of two versions of Morph.

Version (a) improves the evaluation function from the Morph in Version (b) of Figure 0.9. Version (b) adds annealing on top of Version (a).

not rare for Morph to reach the middlegame or sometimes the endgame with equal chances before making a crucial mistake due to lack of appropriate knowledge.

- Morph's database contains many patterns that are recognizable by human players and has given most of these reasonable values. The patterns include mating patterns, mates-in-one, castled king and related defenses and attacks on this position, pawn structures in the center, doubled rooks, developed knights, attacked and/or defended pieces and more.

  We have discovered that adding patterns too frequently, "overloads" Morph so it can't learn proper weights. In the experiment graphed in Figures 0.9 and 0.10, weights are updated after every game but patterns are added only every 7 games. This allows the system to display a steady learning curve as depicted. Morph's games improve based on other metrics as well, such as average game length and TD error (see section 0.2.3).

5.1 Performance Evaluation

To explore new additions to Morph, one implementation is compared with another by using the average number of traditional chess points captured per game as the metric. To be consistent with the experienced-based learning framework, Morph is not made aware of this metric. Each implementation is run until the metric is no longer increasing (Most Morphs stop learning at between 1500 and 2000 games of play). The one with the higher rating is considered the better. We have concluded that only one such comparison is sufficient, because the same version of Morph usually reaches the same average.

5.2 Improvement through Adding New Learning Methods

Adding learning strategies is a gradual process. Each method must be added one at a time to see if it increases performance. If it does than it is kept. Since Morph's initial implementation, such additions have produced significant performance increases. The following two graphs show Morph's cumulative average over time. These graphs compare the performance of four versions of the system. Each version is an extension of the previous one. Figure 0.9a shows a basic Morph, Figure 0.9b shows the result of adding reverse node ordering, Figure 0.10a shows the result of improving the evaluation function, and Figure 0.10b shows the result of adding simulated annealing.

5. Performance Results

To date, Morph has defeated GnuChess twice, and obtained over 40 draws via stalemate, repetition of position and the 50-move rule. The versions of Morph that defeated GnuChess had previously played around 3000 games. However, despite Morph's victory, GnuChess is still much stronger than Morph which is at best a beginning tournament player. In addition to Morph's limited success against GnuChess, there are many encouraging signs since Morph was fully implemented in November, 1990:

- Even though no information about the relative values of the pieces (or that pieces are valuable) have been supplied to the system, after 30 or more games of training Morph's material patterns are consistent and credible (Levinson and Snyder, 1991). (see Table 0.2 for a database after 106 games). The weights correspond very well to the traditional values assigned to those patterns. These results reconfirm other efforts with TD learning (Christensen and Korf, 1986) and perhaps go beyond by providing a finer grain size for material.

Material Pattern					Statistics				Trad.
Pawn	Knight	Bishop	Rook	Queen	Weight	Updates	Variance	Age	Value
0	0	0	0	0	0.455	2485	240.2	106	0
0	0	–1	0	0	0.094	556	7.53	86	–3
0	0	+1	0	0	0.912	653	11.19	88	+3
0	+1	0	0	0	0.910	679	23.59	101	+3
0	–1	0	0	0	0.102	588	17.96	101	–3
0	0	0	–1	0	0.078	667	3.56	103	–5
0	0	0	+1	0	0.916	754	5.74	103	+5
+1	0	0	0	0	0.731	969	22.96	105	+1
–1	0	0	0	0	0.259	861	13.84	105	–1
0	0	0	0	+1	0.903	743	5.68	105	+9
0	0	0	0	–1	0.085	642	3.12	105	–9
0	0	–1	+1	0	0.894	10	0.03	55	+2
0	0	–2	0	0	0.078	146	0.53	71	–6
+1	0	–1	0	0	0.248	26	2.35	73	–2
0	+1	–1	0	0	0.417	81	4.48	82	0
–1	0	–1	0	0	0.081	413	2.14	92	–4
0	–1	+1	0	0	0.478	84	5.72	82	0
+1	0	+1	0	0	0.916	495	3.56	88	+4
0	0	+2	0	0	0.924	168	0.66	91	+6

Table 5.1: A portion of an actual Morph material database after 106 games.

The columns headed by pieces denote relative quantity. The weight column is the learned weight of the pattern in [0,1]. Updates is the number of times that this weight has been changed. Variance is the sum of the weight changes. Age is how many games this pattern has been in the database. Value is the traditional value assigned to this pattern. Note that a weight of 0.5 corresponds to a traditional value of 0. The entire database contained 575 patterns.

- After 50 games of training, Morph begins to play reasonable sequences of opening moves and even the beginning of book variations. This is encouraging because no information about development, center control and king safety have been directly given the system and since neither Morph or GnuChess uses an opening book. It is system (Wilkins, 1980), which, also building on Pitrat's work, used pattern knowledge to guide search in tactical situations. Paradise was able to find combinations as deep as 19-ply. It made liberal use of planning knowledge in the form of a rich set of primitives for reasoning and thus can be characterized as a "semantic approach." This difference, the use of search to check plans and the restriction to tactical positions distinguish it from Morph. Also, Paradise is not a learning program: patterns and planning knowledge are supplied by the programmer. Epstein's Hoyle system (Epstein, 1990) also applies a semantic approach but to multiple simultaneous game domains.

Of course, the novel aspects of APS could not have been achieved without the unique combination of learning methods described here.

4. Relationship to Other Approaches

An APS system combines threads of a variety of machine-learning techniques that have been successful in other settings. To produce this combination, design constraints usually associated with these methods have been relaxed.

The integration of these diverse techniques would not be possible without the uniform, syntactic processing provided by the pattern-weight formulation of search knowledge. To appreciate this, it is useful to understand the similarities and differences between APS and other systems for learning control or problem-solving knowledge. For example, consider Minton's explanation-based Prodigy system (Minton, 1984). The use of explanation-based learning is one similarity: APS specifically creates patterns that are "responsible" (as preconditions) for achieving future favorable or unfavorable patterns through reverse engineering (see Chapter 0.2.2). Also similar is the use of "utility" by the deletion routine to determine if it is worthwhile to continue to store a pattern. The decision is based on the accuracy and significance of the pattern versus matching or retrieval costs. A major difference between the two approaches is the simplicity and uniformity of the APS control structure: no "meta-level control" rules are constructed or used nor are goals or subgoals explicitly reasoned about. Another difference is that actions are never explicitly mentioned in the system. Yee et al. (Yee *et al.*, 1990) have combined explanation-based learning and TD learning in a manner similar to APS. They apply the technique to Tic-Tac-Toe.

It is also interesting to compare APS to other adaptive-game playing systems. Most other systems are given a set of features and asked to determine the weights that go with them. These weights are usually learned through some form of TD learning, with very good success. (Tesauro and Sejnowski, 1989; Tesauro, 1992; Samuel, 1959; Samuel, 1967).

APS extends the TD approaches by exploring and selecting from a very large set of possible features in a manner similar to genetic algorithms. It is also possible to improve on these approaches by using Bayesian learning to determine inter-feature correlation (Lee and Mahajan, 1988).

A small number of AI and machine learning techniques in addition to heuristic search have been applied directly to chess (which, without relying on search, requires much higher precision than backgammon), and then, usually to a small sub-domain. The inductive-learning endgame systems (Michie and Bratko, 1987; Muggleton, 1988) have relied on pre-classified sets of examples or examples that are classified by a complete game-tree search from the given position (Thompson and Roycroft, 1983). The symbolic learning work by Flann (Flann and Dietterich, 1989) has occurred on only a very small sub-domain of chess. The concepts capable of being learned by this system are graphs of two or three nodes in Morph. Such concepts are learned naturally by Morph's generalization mechanism.

Tadepalli's work (Tadepalli, 1989) on hierarchical goal structures for chess is promising. Such high-level strategic understanding may be necessary in the long run to bring Morph beyond an intermediate level (the goal of the current project) to an expert or master level. This brings out both a major weakness and a current topic of research: Morph's current pattern representation scheme is not general and flexible enough for it to create useful plans via EBG. Minton (Minton, 1984), building on Pitrat's work (Pitrat, 1976), applied constraint-based generalization to learning forced mating plans. This method can be viewed as a special case of our pattern creation system. Perhaps the most successful application of AI to chess was Wilkin's Paradise (PAttern Recognition Applied to DIrecting Search)

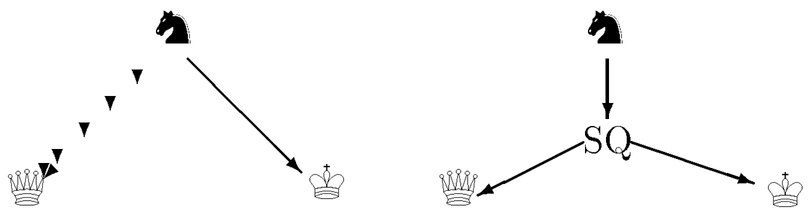

Figure 3.4: The graph on the left depicts a standard pattern derived from node ordering. By applying reverse engineering to the graph on the left we get the graph on the right. The node labeled *SQ* matches squares from which the black knight attacks the white queen and white king simultaneously.

When Morph adds a generalization to the database, it looks for two patterns that have similar weights and then inserts their maximum common subgraph. Specialization occurs when a pattern's weight has a high variance. A new pattern is extracted from the high variance pattern by adding a node and edge to the original graph.

The reverse engineering procedure is performed on each position (state) that occurs in the previous game. For each position, the procedure finds the most extreme pattern p_i that matches it and adds the *precondition pattern* of p_i to the pw database. The precondition pattern, pr_i contains all the nodes in p_i; in addition it may also contain "square nodes" which are nodes that may match any square on the board. To determine the structure of pr_i the system examines each position, starting from the current position and moving towards the initial position, until it finds one in which the structure of the nodes in pr_i differs from that in p_i. pr_i is then extracted from the structure in this earlier position and inserted into the database. An example of a pattern and one of its possible precondition patterns is displayed in Figure 0.8.

Although the genetic operators have not yet been implemented in Morph, mutation and crossover operators are being considered. The mutation operator would be applied each time another addition module was about to insert a pattern. For that pattern there would then be a small probability per node of either flipping its color or changing its piece designation. Finally, the crossover operator would take two extreme patterns and combine half of the nodes in the first with half of the nodes in the second to form a new hybrid structure.

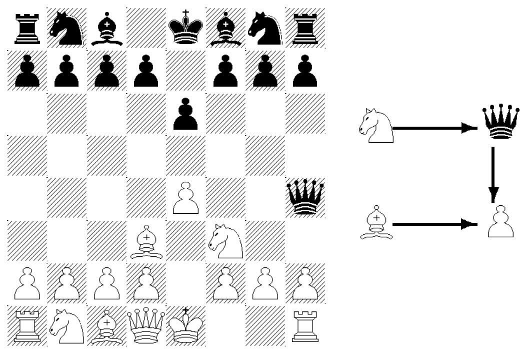

Figure 3.3: In the position shown on the left the pieces no longer on their original squares are moved in the order white pawn, black pawn, white knight, black queen, and white bishop. The subgraph on the right is added when using most recently moved node ordering with a node size of 4. Note, that the black pawn is excluded due to node ordering's preference for the connectivity criterion over the most recently moved criterion.

3.3 Pattern Creation in Morph

This section describes the implementation of the pattern addition modules for graph patterns. As will be seen from the following paragraphs, the implementation is heavily dependent on the pattern representation language. All four types of pattern addition procedures, as described in section 0.2.2, are discussed: search context rules, generalization and specialization, reverse engineering, and genetic operators.

Morph uses two types of search context rules (see section 0.2.2) to add patterns into the database. The search context rules, called node orderings, produce as output a pattern which is a subgraph of the position (state) which is passed in as input. The output pattern is created by numbering the nodes of the position graph according to a node ordering, choosing a random size n, and returning the induced subgraph formed by the first n nodes in the node ordering (and the edges between them). Morph uses two relatively game-independent node ordering rules: In forward node ordering, nodes are ordered by most recently moved piece while maintaining connectivity of the nodes; see Figure 0.7 for an example. In reverse node ordering, nodes are ordered by which piece is next to move while maintaining connectivity of the nodes. In both schemes captured pieces and kings in check are placed high in the list and ties (for squares and unmoved pieces, for instance) are broken randomly. The inclusion of random factors in the above scheme also falls within the genetic algorithm viewpoint, allowing the system to generate and explore a large set of possibilities.

These mathematical constraints to the evaluation function have a strong intuitive backing. For instance, rule 1 states that if two patterns suggest that a position is good ($> .5$) the board should then be considered better than either of them alone. .5 is the weight that is assigned to a pattern that does not have any positive or negative connotation. Patterns with weight 0 suggest strongly that the current board is a losing position and patterns with weight 1 suggest that the current board is a winning position. Table 0.1 shows the application of the evaluation function to several sample values.

Arguments		Result
.5	.5	0.50
.2	.8	0.50
.2	.5	0.20
.8	.5	0.80
.8	.8	0.92
.2	.2	0.08
.1	.8	0.33
.2	.9	0.67
1	.8	1.00
0	.2	0.00

Table 3.1: Results of applying the evaluation function in Morph to various input pairs.

4. if $w_1 < .5$ and $w_2 < .5$ then $f < min(w_1, w_2)$ unless either w_1 or w_2 is 0 then $f = 0$.
5. if $w_1 > .5$ and $w_2 < .5$ then $w_2 < f < w_1$. f is more towards the most extreme weight.

The entire function is displayed in Figure 0.6. This binary function is applied iteratively to the weights of all matching patterns. Note, however, that this function is not associative; thus, the order of evaluation matters. We are currently working on an associative version of this function.

$$f(x,y) = \begin{cases} -.5(2-2x)(2-2y)+1 & \text{if } x \geq \frac{1}{2} \text{ and } y \geq \frac{1}{2} \\ \frac{x-.5+y-.5(2x-1)(2y-1)((2x-1)^2-(2y-1)^2)}{(2x-1)^2(2y-1)^2} & \text{if } x \geq \frac{1}{2} \text{ and } y < \frac{1}{2} \\ f(y,x) & \text{if } x < \frac{1}{2} \text{ and } y \geq \frac{1}{2} \\ 2xy & \text{otherwise} \end{cases}$$

Figure 3.2: Move selection evaluation function.

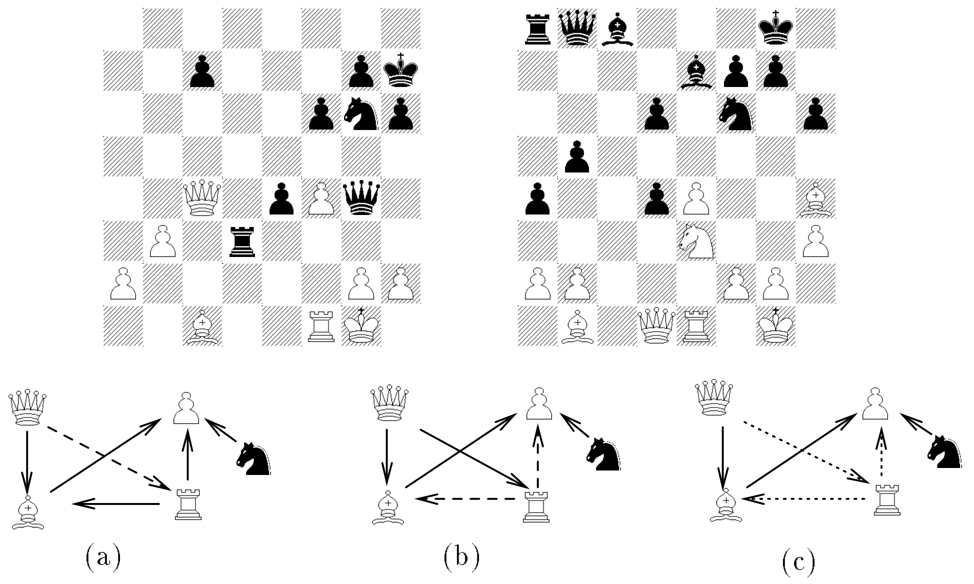

Figure 3.1: A generalization derived from two different chess positions. (a) is the subgraph derived from the board on the left and (b) is the subgraph from the board on the right. The solid edges correspond to direct edges between pieces and the dashed edges correspond to indirect edges. Graph (c) is the generalized graph derived from (a) and (b) in which the dotted edges have been generalized to generic "attacks."

There are actually two types of patterns stored in the Morph system. In addition to the above mentioned graph patterns, Morph stores "material" patterns: vectors that give the relative material difference between the players, e.g. "up 2 pawns and down 1 rook," "even material," etc. Material patterns and graph patterns are processed identically by the system; thus the pw database contains patterns of more than one type.

Along with each pattern is stored a weight in [0,1] as an estimate of the expected true minimax evaluation of states that satisfy the pattern. In order to determine the utility of a pattern and whether it should be retained other statistics about patterns are maintained. These statistic include the number of weight updates the pattern has had, the number of times the pattern has been referenced during play, and the degree to which the weight varies over time.

3.2 Evaluation Function

The evaluation function takes a set of pws matching the position and returns a value in [0,1] that represents an estimate of the expected outcome of the game from that position (0=loss, .5=draw, 1.0=win). It has the following properties, where w_1 and w_2 are the weights of the above mentioned pws.

1. if $w_1 > .5$ and $w_2 > .5$ then $f(w_1, w_2) > max(w_1, w_2)$ unless either w_1 or w_2 is 1 then $f = 1$.
2. if $w_1 = .5$ then $f(w_1, w_2) = w_2$
3. if $w_1 = \alpha$ and $w_2 = 1 - \alpha$ then $f = .5$

3. Morph System

The previous section outlined the framework of a generic APS system. This section describes the actual implementation of one. The APS framework only provides general descriptions for several key elements of the system. These elements include the specific pattern representation, the specific evaluation function and the algorithms used to implement the four pattern addition schemes. This section will describe in detail those decisions that must be made in an actual implementation, in this case, the Morph learning chess program.

Being an APS system, Morph makes a move by generating all legal successors of the current position, evaluating each position using the current pattern database and choosing the most favorable position. After each game patterns are created, deleted and weights are changed to make evaluations more accurate (in the system's view) based on the outcome of the game. Patterns are deleted periodically if they are not considered useful. Morph plays against GnuChess Level I, a program that is stronger than most tournament players.

3.1 Patterns and their Representation

To fully exploit previous experience, the method chosen to represent each experience is critical. An ideal representation is one that uses features that are general enough to occur across many experiences (positions, for chess), but are such that their significance is invariant across these experiences. How to construct such a representation for chess is not obvious. The straightforward approach of using a standard chess board representation is not powerful enough since there are over 10^{40} possible chess positions (Shannon, 1950). In fact, after just three moves for each player, there are over 9 million possible positions (Fox and James, 1987). Further, a pattern such as "white bishop can capture black rook" has nearly the same significance regardless of where on the board the white bishop and black rook are located, and one would like to choose a representation that exploits this information.

In Morph, positions are represented as unique directed graphs in which both nodes and edges are labelled (Levinson, 1991). Nodes are created for all pieces that occur in a position and for all squares that are immediately adjacent to the kings. The nodes are labelled with the type and color of the piece (or square) they represent. For kings and pawns (and also pieces that would otherwise be disconnected from the graph) the exact rank and file on the board in which they occur is also stored. The exact squares of kings and pawns allows the system to generate specific endgame patterns and patterns related to pawn structure. Edges represent attack and defend relationships between pieces and pawns: Direct attack, indirect attack, or discovered attack (a defense is simply an attack on one's own piece). At most one directed edge is assigned from one node to another and the graph is oriented with black to move. Patterns come from subgraphs of position graphs (see Figure 0.5) and hence are represented the same way except that they may label any node with an exact or partial square designation. The representation has recently been extended to include other squares as nodes besides those adjacent to kings.

A similar representation scheme has successfully been used to represent chess generalizations (Zobrist and Carlson, 1973) and to produce a similarity metric for chess positions (Levinson, 1991; Botvinnik, 1984).

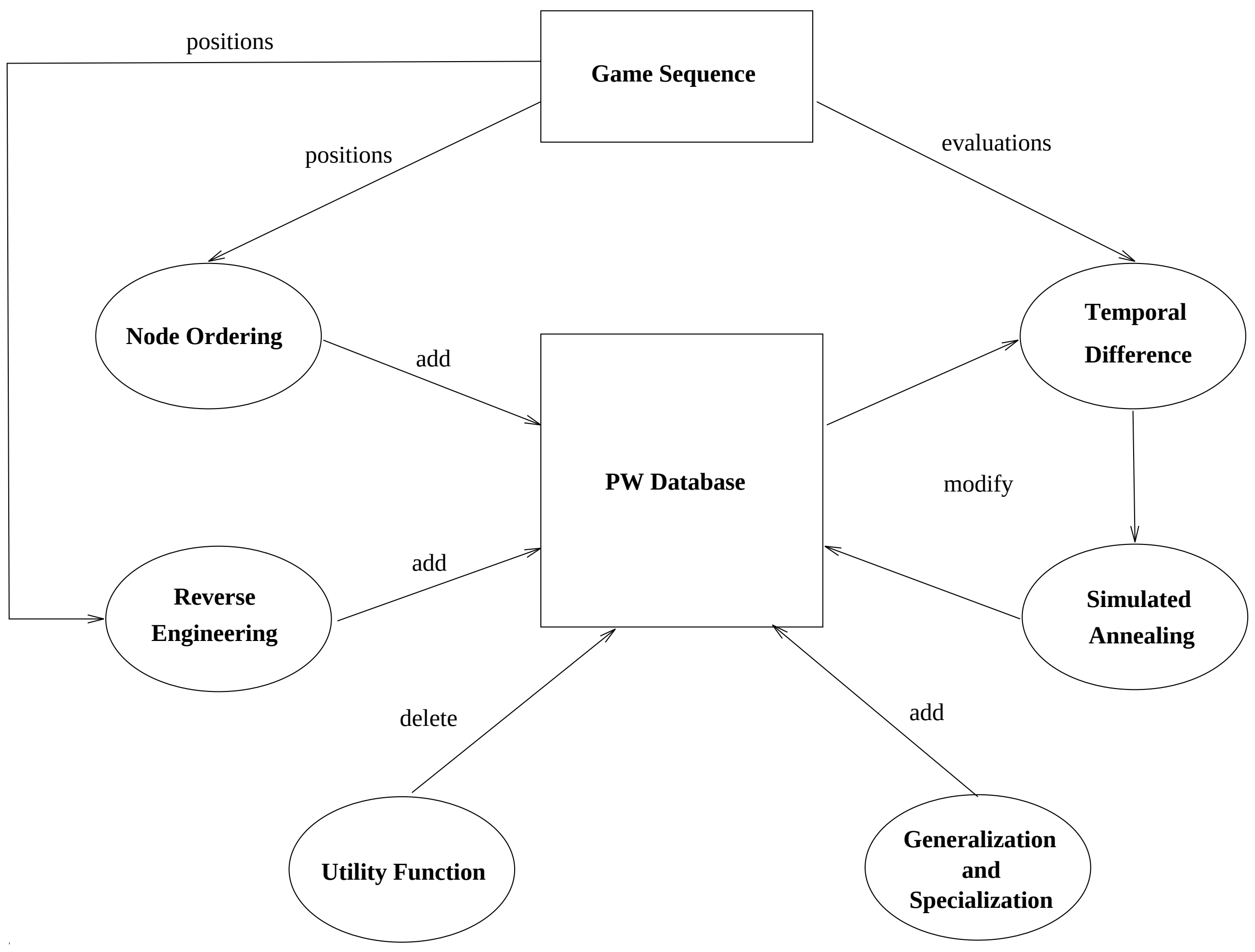

Figure 2.4: The integration of learning modules within an APS system.

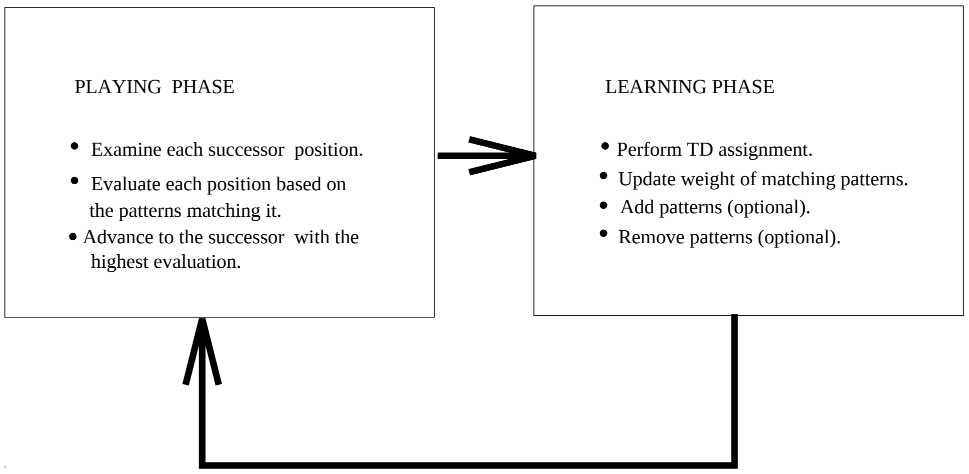
PLAYING PHASE
• Examine each successor position.
• Evaluate each position based on the patterns matching it.
• Advance to the successor with the highest evaluation.
LEARNING PHASE
• Perform TD assignment.
• Update weight of matching patterns.
• Add patterns (optional).
• Remove patterns (optional).

Figure 2.3: The execution cycle of an APS system.

The search phase traverses a path from the initial state of the problem domain to a reinforcement state. Depth first hill climbing is the search technique used to traverse the search tree. In other words, at each state S_i in the search path the state moved into next, S_j, is determined by applying an evaluation function to each successor state of S_i and choosing that state which has the highest evaluation. Note, however, nothing prevents the database from supporting more sophisticated search strategies.

Although the exact calculations performed by the evaluation function depend on the particular APS system, the function has the following framework. It takes the state to be evaluated and determines the most specific pws in the pw database that match that state. The weights of these most specific pws are then combined by a system dependent rule to determine the final evaluation for that state.

The learning phase, the second in the APS cycle of execution, takes as input the sequence of states traversed in the search performed by the first phase. This sequence is used by TD learning and simulated annealing to modify the weights of the patterns in the database. Patterns are then inserted into the database using the four techniques mentioned in Section 0.2.2. Finally, unimportant patterns are removed from the database as described in the same section.

The execution of an APS system involves the interaction of many learning techniques. A global view of this interaction is displayed in Figure 0.4. In this figure the edges are labelled with actions specifying whether a given module adds patterns, deletes patterns or modifies the weights of patterns. Central to the APS system is the pattern weight database, which holds all of the accumulated search knowledge generated and manipulated by the surrounding modules.

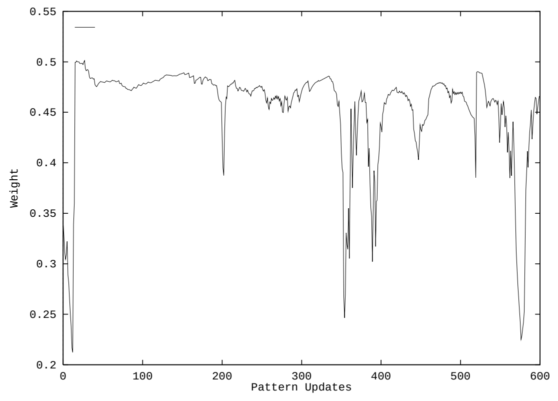

Figure 2.1: This graph depicts the weight change of the material pattern "down 1 bishop" in a system without simulated annealing.

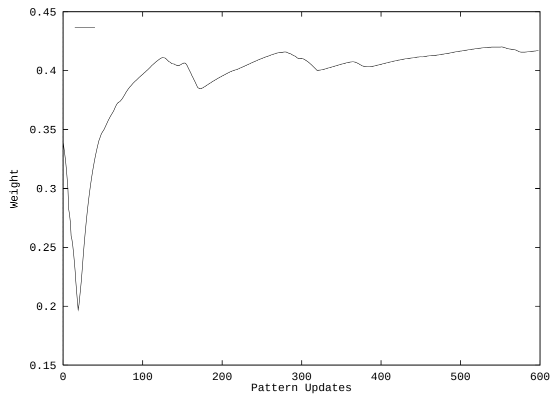

Figure 2.2: This graph depicts the weight change of the same pattern as Figure 0.1 in a system that has simulated annealing added.

2.5 Integration: The workings of the entire system

This section describes the workings of a complete APS system by combining the parts described in the previous three sections. An APS system executes by repeatedly cycling through two phases: a search phase and a learning phase (see Figure 0.3). In the search phase the APS system does not modify its knowledge base but instead performs a search using the current knowledge base. The learning phase then alters the knowledge base by adding and deleting patterns, and modifying the weights of patterns.

- Moving a physical system to its lowest energy configuration is then analogous to finding the configuration that optimizes the objective function.
- The configuration of the system is then the different informational parameters that the system can have.
- The temperature is a mechanism for changing the parameters.

In APS, the system's situation is similar to that of the statistical physicist. The database is comprised of a complex system of many particles (patterns). The goal is to reach an optimal configuration, i.e. one in which each weight has its proper value. The average error (i.e. the difference between APS's prediction of a state's value and that provided by temporal-difference learning) serves as the objective evaluation function. Intuitively, the average error is an acceptable performance evaluation metric if one accepts the empirical and analytical arguments that TD learning is a convergent process (Sutton, 1988): since TD learning will produce evaluations close to the true ones the error will be high or low depending on the APS system's degree of accuracy. Indeed, experimentally we have observed a high correlation between TD error and more domain-specific performance metrics.

The configuration of an APS system is made up of its pws. Temperature corresponds to the rate at which each weight moves towards the value recommended by TD learning. In addition to using a global temperature that applies to all weights, each weight has its own local temperature. This is done to give each pw its own learning curve that depends on the number of states it has occurred in. A pw that occurs in many states has its temperature reduced more quickly than a pw that occurs in only a few states, because the first pattern has more examples to learn from and hence early convergence is appropriate. Each pw's learning rate is affected by its number of updates and the global temperature as follows:

$$\text{Weight}_n = \frac{\text{Weight}_{n-1} * (n-1) + k * \text{new}}{n + k - 1}$$

$Weight_i$ is the weight after the ith update, new is what TD recommends that the weight should be, n is the number of the current update and k is the global temperature. When $k = 0$ the system only considers previous experience, when $k = 1$ the system averages all experience, and when $k > 1$ the present recommendation is weighted more heavily than previous experience. Thus, raising k creates faster moving patterns. As an example of how global temperature affects the learning rate, if the system doubles the global temperature, a pattern that has 100 updates will be updated like a pattern that has 50 updates the next time its weight is modified.

The above discussion describes how simulated annealing gives each pattern its own learning rate. The figures below, taken from the Morph system, demonstrate the effectiveness simulated annealing has in forcing convergence of the weights in the system. Figure 0.1 and Figure 0.2 show the weight change over time for the material pattern "down one bishop." The first figure comes from a version of Morph that does not use simulated annealing; the weight fluctuates within a certain range. The second figure displays the same pattern from a Morph with simulated annealing. Here, the weight homes in on a specific value.

grown to 2500 patterns the Morph chess system takes twice as long to play and learn (from about 600 games a day on a Sun Sparc II to about 300 games per day).

2.3 Modifying the Weights of Patterns

The modification of weights (of patterns) to more appropriate values occurs every time a reinforcement value is received from the environment. The modification process can be broken down into two parts. First, each state in the sequence of states that preceded the reinforcement value is assigned a new value using temporal difference learning. Second, the new value assigned to each state is propagated down to the patterns which matched that state. A pattern in the database matches a given state if the state satisfies the boolean feature represented by the pattern and no other pattern more specific than it matches the state.

TD learning determines new values for the states in the game sequence moving from the last state, S_n, to the first state, S_1. Since state S_n was the state at which reinforcement was delivered its new value is set to the reinforcement value. For all other states, S_i, the new value is set to a linear combination of its old value and the new value of state S_{i+1}. This method differs from supervised learning, where the value of each state S_i is moved toward the reinforcement value. It has been shown that TD learning converges faster than the supervised method for Markov model learning (Sutton, 1988). The success of TD learning stems from the fact that value adjustments are localized. Thus, if the system makes decisions all the way through the search and makes a mistake toward the end, the earlier decisions are not harshly penalized.

Once each state S_i has been assigned its new value, each pattern matching S_i must have its weight moved towards the new value. Each weight is moved an amount relative to the contribution the pw had in determining the old value for the state.

This weight updating procedure differs from those traditionally used with TD learning in two ways. First, TD learning systems typically use a fixed set of features, whereas in an APS system, the feature set changes throughout the learning process. Second, the APS system uses a simulated annealing type scheme to give the weight of each pattern its own learning rate. In this scheme, the more a pattern gets updated, the slower its learning rate becomes. Furthermore, in addition to giving each pattern its own learning rate, the annealing scheme forces the convergence of patterns to reasonable values.

2.4 Simulated Annealing

Simulated annealing is a learning procedure that has been derived from a practice in statistical mechanics (Davis and Steenstrup, 1987). In this area it is often desirable to take a system of molecules and reduce it to the lowest possible energy by lowering temperature. Through experience it has been found that if the temperature is reduced too quickly the system will have a small probability of being at an optimally low temperature. Metropolis et al. (Metropolis *et al.*, 1953) developed a technique for lowering the temperature gradually to produce (on the average) very low energy systems at the lowest temperature.

Kirkpatrick et al (Kirkpatrick *et al.*, 1983) adapted annealing to computer science, by finding information analogies for their physical counterparts (Davis and Steenstrup, 1987):

- The energy of the system becomes an objective function that describes the quality of the current system configuration.

make a further distinction. At other times it can be simplified (generalized) without loss of discriminative power.

Generalization patterns are created by extracting similar structures from within two patterns that have similar weights. A pattern is specialized in an APS system if its weight must be updated a large amount (indicating inaccuracy). Whereas in a standard concept induction scheme the more specific patterns may be deleted, the APS system keeps them around because they can lead to further important distinctions. The deletion module may delete them later, if they no longer prove useful.

Reverse engineering is a method used to add macro knowledge into an APS system. Macros can be represented as pws by constructing a sequence of them such that each pattern is a precondition of the following one. Successive weights in the macro sequence will gradually approach a favorable reinforcement; thus, the system is then motivated to move in this direction.

Reverse engineering extends a macro sequence by adding a pattern. This extension is similar to Explanation-Based-Generalization (EBG) (Mitchell *et al.*, 1986b) or goal regression. The idea is to take the most important pattern in one state, s_1, and back it up to get its preconditions in the preceding state, s_2. These preconditions then form a new pattern p_2. If pattern p_2 is the most useful pattern in state s_2, it will be backed up as well, creating a third pattern in the sequence, etc. The advantages of this technique are more than just learning "one more macro"; each of the patterns can be used to improve the evaluation of many future positions and/or to enter the macro at any point in the macro sequence.

Genetic algorithms (Holland, 1975) are a means of optimizing global and local search (Glover, 1987). In these algorithms, solutions to a problem are encoded in bit strings that are made to evolve over time into better solutions in a manner analogous to the evolution of species of animals. The bits in the solution representation are analogous to genes and the entire solution to a chromosome. In such systems there are a fixed number of solutions at any one time (a generation). Members of each generation interbreed to form the next generation. Each genetic algorithm has a fitness function that rates the quality of the solution in a particular generation. When it is time for a new generation to be created from the current generation, the solutions that are more fit are allowed to be the more active breeders. Breeding usually involves three processes: crossover, inversion, and mutation.

The APS system makes use of genetic operators in order to add additional patterns into the database. Since patterns are not required to be represented as bit strings (in fact in Morph they are graphs), it is up to the individual APS system to tailor the genetic operators to suit the pattern representation. The APS system does not remove all or most of a population of patterns, however, due to the large amount of time necessary in determining appropriate weights for all patterns.

Although a variety of pattern addition schemes are available, due to memory and processing restrictions, the database must be limited in size. As in genetic algorithms there must be a mechanism for insignificant, incorrect or redundant patterns to be deleted (forgotten) by the system. A pattern should contribute to making the evaluations of states it is part of more accurate. The utility of a pattern can be measured as a function of many factors including age, number of updates, uses, size, extremeness and variance. These attributes will be elaborated upon in the next section. We are exploring a variety of utility functions (Minton, 1984). Using such functions, patterns below a certain level of utility can be deleted. Deletion is also necessary for efficiency considerations: the larger the database, the slower the system learns. For example, after 2000 games of training and with a database

2. The APS Model

As mentioned previously, the APS framework contains three major parts: the pattern-weight formulation of search knowledge, methods for creating and removing pws, and methods for obtaining appropriate weights for the pws with respect to reinforcement values. This section discusses each of these facets in detail, after which it describes how the parts interact and how the system performs as a whole.

2.1 Pattern weight formulation of search knowledge

A pattern represents a boolean feature of a state in the state space. This feature typically represents a proper subset of all the possible properties of the state. That is the feature usually does not represent a single state, because such patterns would be far too specific (and numerous) to be useful in complex problem domains. Examples of patterns include graphs and sets of attributes. Often the language in which patterns are expressed is akin to the language used to represent the states, but with a higher level of abstraction, e.g., see discussion of Morph below.

Each pattern has associated with it a weight that is a real number within the reinforcement value range. The weight denotes an expected value of reinforcement, given that the current state satisfies the pattern. For example, in a game problem domain a typical set of reinforcement values is $\{0,1\}$, for loss and win respectively. If we have a pw, $<p_1, .7>$, this implies that states which have p_1 as a feature are more likely to lead to a win than a loss.

The major reason for using pws over another form of knowledge representation is their uniformity. Pws can simulate other forms of search control and due to their low level of granularity and uniformity more power and flexibility is possible (Levinson, 1989b). For example, pws have all the expressive power of macro tables. Additionally, they allow switching over from one macro sequence to another and allow for the combination of two or more macro tables(Levinson, 1989b).

2.2 Adding and removing patterns

The patterns used to represent search knowledge are stored within a database that is organized as a partial order on the relation "more-general-than". Patterns are inserted into this database through the following four methods: search context rules, generalization and specialization, reverse engineering, and genetic operators.

Search context rules are the only pattern addition scheme that does not rely on patterns already in the database; thus, they are the only way patterns are added to an empty database. A search context rule takes as input a particular state and the sequence of all states in the last search and returns a pattern to be inserted into the database. A search context rule is just a deterministic procedure that builds up a pattern given the previously mentioned inputs. Examples of search context rules can be found in Section 0.3.3.

In concept induction schemes (Michalski, 1983; Mitchell *et al.*, 1986a; Niblett and Shapiro, 1981; Quinlan, 1986) the goal is to find a concept description to correctly classify a set of positive and negative examples. In general, the smaller description that does the job, the better. Sometimes the concept description needs to be made more specific to

1. Introduction

This paper introduces *experience-based learning.* This term refers to that type of unsupervised reinforcement learning in which almost all responsibility for the learning process is given to the system. These responsibilities include state evaluation, operator (move) selection, feature discovery and feature significance. As a learning framework experience-based learning can be applied to many problem domains (Levinson *et al.*, 1992).

The types of problem domains considered here are restricted to complex problem domains characterized by three features. First the problem must have a formulation as a state space search. Further, reinforcement is only provided occasionally, and for many problems only at the end of a given search. Finally, the cardinality of the state space must be sufficiently large so that attempting to store all states is impractical.

This paper describes a learning system architecture called an *adaptive predictive search* (APS) system (Levinson, 1989a; Levinson *et al.*, 1992), which handles experienced-based learning in complex problem domains. The first key aspect of an APS system is the compilation of search knowledge in the form of pattern-weight pairs (pws). Patterns represent features of states, and weights indicate their significance with respect to expected reinforcement. Secondly, since the APS system resides within the experience-based learning framework, it must possess facilities for creating and removing search knowledge (pws). Knowledge is maintained to maximize the system's performance given space and time constraints. The APS model uses a variety of techniques for inserting and deleting patterns. Finally, a combination of several learning techniques incrementally assign appropriate weights to the patterns in the database. The specific insertion, deletion, and learning techniques will be described in the next section.

Since APS adheres to the experience-based learning framework, it can be applied to new domains without requiring the programmer to be an expert in the domain. In fact, APS has been applied to a variety of domains including chess, othello, pente, and image alignment (Levinson *et al.*, 1992). The chess implementation, Morph, is also described in this paper.

Chess satisfies all of the abovementioned requirements of the complex problem domains considered in this paper: The game tree forms the state space, each game representing a search path through the space; reinforcement is only provided at the end of the game; and, finally, it has a large cardinality of states (around 10^{40}) (Shannon, 1950). Furthermore, few efforts use previous search experience in this area despite the high costs of search. In order to focus the research on the learning mechanisms, Morph has been constrained to using only one-ply of search. In addition, Morph has been given little initial chess knowledge, thus, keeping it within the experienced based learning framework. Despite these constraints, Morph has managed several dozen draws and two wins against Gnuchess, a traditional search based chess program stronger than most tournament players.

The structure of the rest of the paper is as follows. The next section discusses the APS system framework in detail. In Section 3 the Morph APS implementation is described. Section 4 compares APS to other efforts in adaptive search and adaptive game-playing systems. This is followed by performance results in Section 5 and, finally, a conclusion in which future research and open questions are discussed.

Experience-Based Adaptive Search

Jeffrey Gould and Robert Levinson
Department of Computer and Information Sciences
University of California Santa Cruz
Santa Cruz, CA 95064 U.S.A
(408)-458-9792 (408)-459-2565
jeffg@cis.ucsc.edu levinson@cis.ucsc.edu

UCSC-CRL-92-10
April 16, 1992

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA