This paper appeared in *Journal of the Learning Sciences*, 1993, 4(3): 281-319

Learning in the Right Places

Susan L. Epstein

Hunter College and The Graduate School of The City University of New York

Abstract

A learner's experience is in large measure determined by the situations it encounters in the problem space, and, for a challenging task, only a small fraction of that space can ever be visited. People's ability to learn to perform well at such tasks is therefore a clear indication that not all situations are equally relevant to learning. The primary contributions of this paper are the notion of *key nodes*, situations particularly important to the development of expertise, the presentation of empirical evidence for their existence, and the design of a training method intended to capitalize upon them. The paper also suggests why key nodes might be clustered in the search space, and describes how a learner might arrive at a key node by chance, be drawn there by a choice the learner makes, or be driven there by the actions of another agent. This paper offers empirical evidence of substantial improvement in the quality of performance when a game-learning program is deliberately directed to clusters of key nodes, and considers several ways to do so. It also discusses the extension of these results to other domains, and speculates on the significance of these results for human learners.

1. Introduction

This paper addresses the acquisition of strategic knowledge that supports expert behavior. The thesis of this work is that expert reasoners can learn to perform difficult tasks well because they identify relatively few decision-making points as key and then extract crucial knowledge from them. We hypothesize that the existence of *key nodes*, as we call these information-rich points, permits a learner to extract enough knowledge to perform like an expert without experiencing all possible decision-making situations. Key nodes enable a reasoner to parlay a learning experience that is small in comparison to all possible situations into a robust expertise that can succeed under any circumstances. If this premise is correct, then the quality of learned performance depends in part upon the learner's ability to locate key nodes. In other words, a good learner knows not only what and how to learn, but also which situations offer important learning experiences.

Informally, key nodes offer more of the information required to perform expertly than other problem situations do. They are also situations where the learner's decision would have been uninformed; all the alternatives might have seemed poor, the correct choice might have seemed among the weakest, or the learner might have had no clear preference. Insufficient knowledge to make a well-founded decision can motivate a reasoner to learn. The claim here is that the same situations which trigger this need to learn also act as key nodes, because examination of them offers knowledge pertinent to expert performance.

Consider, for example, a driver with a formula that directs route selection in an unmapped town of many one-way streets and dead ends. That driver will be considered an expert once its formula is better than most people's (D'Andrade, 1991). The development of that formula, however, may require steering the car to key nodes, places where the driver discovers information that improves its formula. A dead end is an example of a key node. The driver tries a route that includes the dead end and learns that it offers no ready access to most locations in town. The motivation for visiting the dead end the first time was not to learn about every street in town (knowledge about the task), but to find an efficient and effective route (knowledge about how to behave there). A second example of a key node is a traffic circle that offers along its short circumference a convenient interchange among eight major thoroughfares. Note that key nodes may (the traffic circle) or may not (the dead end) be places worth revisiting.

The terminology of state spaces clarifies the substantial difference between knowledge about a task and knowledge about how to perform it. A *state* is a description of some situation where a decision must be made, such as a location where a driver may turn left, turn right, or continue straight ahead. A *space* is the set of all possible states that can arise in a particular task, such as all possible locations in a town that require a driving decision. The states in a space are linked together by the permissible decisions there. In the driving example, a decision at one place will lead to some subsequent location where a choice is once again required. Thus a location with three options (left, right, straight) is linked to each of the three subsequent states encountered when one of those options is chosen. In the driving example, knowledge about the task is knowledge about the states that exist and how they are linked together, while knowledge about how to perform the task is knowledge about how to drive from one point to another. In this state-space representation, *problem solving* is navigation through a sequence of states in the space, and *expertise* is correct and efficient navigation. A set of related problem spaces, like driving through unmapped towns or playing board games, is called a *domain*.

Well-guided search of a state-space thus becomes an appropriate metaphor for well-directed learning. Well-guided search describes a learning experience in which the learner arrives at the key nodes, the places where knowledge to support behavioral expertise can be acquired. The central issues in well-guided search concern the balance among learning speed, learning accuracy, and the development of self-reliance in the learner. In the context of key nodes, these issues become proof that key nodes exist, methods to reach them, and their role in the learning process. Does how to encounter key nodes vary with the problem class, for example with the town in the driving example? Does how to encounter key nodes vary with the problem domain, for example, with driving as opposed to flying? How does one learn to encounter key nodes efficiently? To what extent does knowledge about how to perform expertly overlap with knowledge about how to encounter key nodes? To what extent does the learning method for one inform the learning method for the other?

The premise that key nodes exist arose during experiments with *Hoyle*, a program that learns to play two-person, perfect information, finite-board games. Hoyle begins with general game-playing knowledge plus the rules for one specific game, and then learns during competition to play that particular game better. Competition may be against an *external* contestant (a person or another program) or Hoyle itself. As Hoyle analyzes its experience and identifies key decision-making situations, the program extracts game-specific knowledge it can apply in later competition at the same game. To date Hoyle has learned to play 18 different games as well as human experts.

This paper details aspects of Hoyle's behavior for which only the existence of key nodes provides an explanation. Guidance to key nodes, we show, results from both *internal direction* (from the learner's reasoning principles) and *external direction* (from the learner's environment). The results argue for carefully designed teaching environments that value both instruction and practice; they also recognize certain aspects of an individual that make learning possible. The empirical results described here are from the domain of board games, but broader general applicability is discussed as well. The contributions of this paper are the identification of key nodes and well-guided search as important concepts, the demonstration of their power, and the formulation of an initial theory for them.

The next section consists of some simple definitions to establish common terminology. Subsequent sections explain the role of key nodes in experience, provide background on Hoyle, and document how a learner can arrive at key nodes through both external and internal direction. The paper goes on to discuss these results in the broader context of cognitive science and related work. The final section outlines the beginnings of a theory for well-guided search. Descriptions and representations for all referenced games appear in the Appendix.

2. Some Fundamental Definitions

Game playing is a good domain in which to study key nodes. Each state can be completely described by relatively few values. Games are noise free, that is, descriptions are always correct, without intervening instrumental or human error. It is easy to replicate a situation where a decision must be made, or even a sequence of such situations. For simple games, and even for many states in more difficult games, the correct decision is readily computable. Finally, it is easy to measure different facets of expertise as performance in competition against a variety of opposition.

Although human experts for games like chess certainly exist, little is known about *how* people learn to become experts. Psychologists have studied expert human game players, particularly chess players, for a century, but it is the nature of their skill, rather than its acquisition, that has been the primary focus of attention (Binet, 1894; Charness, 1981; Djakow, Petrowski, & Rudik, 1927; Holding, 1985). When the study of expertise is extended to other domains, the focus has been primarily on experts' memories and their organization (Allard, Graham, & Paarsalu, 1980; Chase & Simon, 1973; Egan & Schwartz, 1979; Eisenstadt & Kareev, 1975; Engle & Bukstel, 1978; Goldin, 1978; Shneiderman, 1976; Watkins, Schwartz, & Lane, 1984). Work on the transition from novice to expert has focused primarily on the reorganization of knowledge as in (Chi, Feltovich, & Glaser, 1981). An exception to this is recent work on the acquisition of tic-tac-toe (or naughts and crosses) skill by six- to nine-year-old children, who are shown to acquire strategies such as Win, Fork, and Block with age and, presumably, with more experience at the game (Crowley & Siegler, 1993).

The importance of well-guided search is supported by the fact that "most of the recent world champions in chess were at one time tutored by chess masters" (Ericsson & Charness, 1994, p.739). Indeed, extensive studies across many domains support the hypothesis that *deliberate practice* (activities found to improve performance most effectively) is fundamental to the development of expertise (Ericsson, Krampe, & Tesch-Römer, 1993). This paper addresses the nature of the situations that deliberate practice exposes.

There is only one program, to the best of our knowledge, that has strong connections with human game playing and game learning. Hoyle, described further in Section 4, learns to play 18 different games expertly, and uses multiple learning methods to do so. The program employs commonsense strategies like those Crowley and Siegler have detected in people, and does not insist that its knowledge be absolutely correct. Also like people, Hoyle's learning algorithms are selective enough to control the amount of knowledge it acquires, and it makes use of visual symmetries. Many of the program's learning methods are familiar even to a novice human game player. Hoyle can express the reasons for its move selection in terms people understand. Using learning time as a measure of difficulty, Hoyle's performance orders three games identically to the way seven human subjects' performance does (Ratterman & Epstein, in preparation). Hoyle is a discovery program, that is, like a person learning to play, it shapes much of its own learning experience by its behavior. Hoyle is the program whose empirical data instigated the theory of key nodes described here.

The following definitions facilitate the discussion of game playing in the remainder of this paper. Some of the ideas are illustrated in Figure 1.

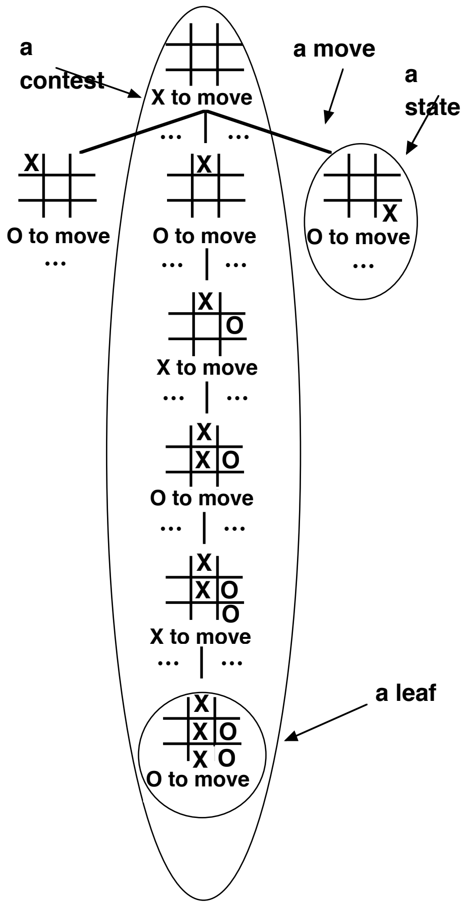

Figure 1. A portion of the tic-tac-toe game tree.

• A *game* is an activity for two agents, called *contestants*. Each contestant is assumed here to have *perfect information*, access to all the relevant knowledge about the current state of the world, so that there are no concealed cards, for example. A game is defined by a finite board, playing pieces, and a set of rules determining play. For example, the rules of tic-tac-toe stipulate that the grid is initially empty, that the contestants alternate placing an owned playing piece in any empty position, and that play halts if the grid is filled or if one contestant (the winner) achieves three owned pieces along a row, column, or diagonal.

• A *position* is a location on the game board where a playing piece may rest, in accordance with the rules of the game. For example, there are nine positions on the tic-tac-toe board.

• A *state* is a configuration of playing pieces on the board along with the identity of the *mover*, the contestant whose turn it is to play. A *reachable state* is one that can actually occur during competition. For example, an empty grid with X to move is a reachable tic-tac-toe state, but with O to move it is not. Interesting board games usually have a great many reachable states.

• A *move* is an action that is permitted by the rules of the game and transforms one state into another. For example, from the empty grid state with X to move in tic-tac-toe, there are nine legal moves, each of which results in a different grid with a single X, eight empty positions, and O to move.

• A *game tree* organizes the set of all reachable states for a game so that each state points to all possible next states after a single legal move. For example, in the tic-tac-toe game tree the initial empty-grid state points to its nine next states. A game tree is the *search space* (set of possible problem situations) for a problem-solving program that plays a game. The average number of states pointed to by any single state in a game tree is called its *average branching factor*. This is a metric that reflects the static nature of the game tree, and is unrelated to any particular search strategy.

• A *contest* is one complete experience at a game, from some initial state specified by the rules to some final state where the rules designate a winner or declare a draw. A contest may be envisioned as a path through a game tree from the initial state to a *leaf*, a state which points to no others. The first moves of a contest are called the *opening*, the last moves the *endgame*, and the rest is referred to as the *middlegame*.

• A *tournament* is a sequence of contests between two contestants in which they alternate moving first. One might, for example, play a tournament of 20 contests at the game of tic-tac-toe.

• *Perfect play* is when, for every state in the game tree, one always moves to secure at least the best possible outcome despite subsequent error-free play by the opposition. With perfect play one always wins from a winnable position and draws from a drawable one.

• A *draw game* is one in which every contest between two perfect contestants must, by the nature of the game graph, end in a draw. Tic-tac-toe is an example of a draw game.

• A contestant who loses a contest at a draw game has made an error; one who wins or draws is said to have *achieved an expert outcome*.

3. Key Nodes

Given a learning program with competence in more than one problem space, key nodes by definition exist in every problem space where that program learns from experience to perform better. We begin with a formal definition for key nodes:

For any search space S, the key nodes K for a learner L with foreknowledge F are a minimal subset of nodes from S such that when L begins with F, learns while it visits K, and then turns learning off, L performs expertly throughout S.

The remainder of this section first looks more carefully at the role of the foreknowledge F and the minimality restriction, and then offers evidence of the existence of key nodes and discusses their relation to learning. Several examples of key nodes in game playing are included.

3.1 The Role of Foreknowledge in Key Nodes

The property of being a key node is a function of the learner's foreknowledge, that is, what one knows in advance necessarily delimits what a key node is. Consider first the situation in Figure 2(a). Most people who consider themselves expert at tic-tac-toe believe that O's prospects look grim here. Not coincidentally, this is a key node (an element of K) in the game of tic-tac-toe (S) for Hoyle (L) whose general game-playing foreknowledge (F) is detailed in Section 4. Tic-tac-toe is so easy that if Hoyle encounters this situation once in play against an external expert, it will acquire all the additional knowledge beyond F that it needs to play tic-tac-toe perfectly. If the program does not encounter Figure 2(a) or its symmetric equivalent, and learning is turned off, Hoyle will never play tic-tac-toe perfectly, because it will always make the wrong move (into a corner) here.

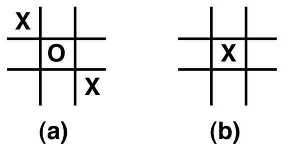

Figure 2. Key nodes for Hoyle with O to move against an expert as X in the search space (a) for tic-tac-toe and (b) for lose tic-tac-toe.

As another example, consider Figure 2(b), a key node (an element of K) in the game of lose tic-tac-toe (S) for Hoyle (L) with the same general game-playing foreknowledge (F). Like tic-tac-toe, lose tic-tac-toe is played on a 3 ∞ 3 grid between X and O. Whoever achieves three owned playing pieces in a row, however, vertically, horizontally, or diagonally *loses*. Error-free lose tic-tac-toe contests are always draws. This game is harder for Hoyle to learn than tic-tac-toe. Most people who have never played this game before are certain that X has made a serious error in Figure 2(b). Hoyle, too, will persistently try every possible way to play as X from the other openings; its foreknowledge incorrectly makes Figure 2(b) unattractive. If Hoyle is playing X the first time it experiences Figure 2(b), its foreknowledge and whatever it has learned thus far are likely to be so inadequate that it still loses, despite happening upon the correct opening. The program's learning from that failure could further discourage use of the correct opening in the future. If Hoyle is playing O against an external expert when it encounters Figure 2(b), however, it will learn the opening and go on to use it in subsequent contests.

3.2 The Minimality of Key Nodes

Because they are a *minimal* subset of S, key nodes offer an efficient path to important knowledge. In Figure 2(b), after all possible error-ridden contests with the other openings are experienced, the program can deduce the knowledge that would have been supplied immediately by the key node. In the course of this experience, Hoyle would acquire extensive, detailed knowledge about each of the alternative openings, much of it irrelevant to expertise at lose tic-tac-toe. Unfortunately, when the other contestant realistically and deliberately varies its play, happening upon every subsequent way to lose is likely to require hundreds or even thousands of contests. Even if the other contestant were to deliberately instruct Hoyle by playing in turn each possible lose tic-tac-toe contest with the wrong opening, learning would require more than a hundred contests. How or why one loses with the other openings is unimportant to behavioral expertise in a game that always ends in a draw with error-free play on both sides. The efficient summary available from Figure 2(b) is “open in the center.”

3.3 Evidence of Key Nodes’ Existence

Tic-tac-toe offers a simple, well-understood example of the contrast between the total number of situations an expert might theoretically confront and the number of situations a developing expert experiences during learning. Although initially the combinatorics of tic-tac-toe are a bit daunting, Table 1 shows that the rules of the game reduce the number of situations that actually arise during play to about 5478. People do not need to experience all 5478 to develop expertise; in several games of this size, human subjects often achieve expertise after exposure to at most 7% of the possible situations (Ratterman, et al., in preparation). There are also some basic symmetries in tic-tac-toe, such as “there is no difference between beginning with the first X in the upper left corner or in the upper right.” These symmetries reduce the possible situations even further; for example, 9 possible first moves become 3. Even after reduction for symmetry, however, the 7% encountered during learning do not encompass all possible situations. Perhaps experts learn sequences of responses rather than what to do in every possible situation. There are, after all, only 170 different ways to play an entire tic-tac-toe contest if one recognizes symmetrically equivalent situations, applies them to narrow the possibilities, and stores sequences of up to nine moves (calculated from Berlekamp, Conway, & Guy, 1982, pp.670-671). Few human experts, however, experience all 170 scenarios, let alone remember them.

Table 1: Reducing combinatoric complexity in tic-tac-toe

	RepresentationCount
States based on average branching factor and contest length	$9^{4.5} = 19{,}683$
States calculated from combinatoric formulae	6046
States that actually occur during rule-abiding play	5478
Contests that sequence symmetrically-distinct occurring states	170

Key nodes appear in much larger spaces as well. Data indicate striking similarities between the way ordinary folk cope with the somewhat oversized tic-tac-toe search space and the way grandmasters at chess cope with an enormous one. The literature on expert chess players confirms that they neither search nor remember very much of such a space. With an average branching factor of about 20 and contests that average 80 moves, the chess game graph could have as many as $20^{80} \approx 10^{120}$ nodes. Although there may be as many as 40,000 distinct chess openings, most tournament players encounter no more than 400 of them in the course of a year of play, and prepare only three or four for any given tournament (Charness, 1991). If one restricts the graph to these 40,000 openings, assumes a memorized endgame, and restricts moves to those that masters might select, there would still be $4 \infty 10^{20}$ reasonable contests with exhaustive search (Charness, 1983). An expert chess player is estimated to experience 1,200,000 nodes per year (Charness, 1991). During 20 years of learning, that amounts to only about $6 \infty 10^{-14}$ of such a reasonable space, and $2.4 \infty 10^{-113}$ of the entire space.

The famous chess-playing computers, like HiTech and Deep Thought, experience more nodes in a minute than any grandmaster does in a year (Anantharaman, Campbell, & Hsu, 1990; Berliner & Ebeling, 1989). Although the programs outplay most skilled human chess players, there are still hundreds of grandmasters who consistently outplay the machines. Those people *learned* to play that well, better than any high-speed, deep-searching chess program, in only a tiny fraction of the chess space. This suggests that the chess game tree, like the tic-tac-toe game tree, does not uniformly distribute the knowledge necessary for people to learn to play well across all its states, that is, that some of the visited nodes trigger important learning experiences and that learners regularly encounter those nodes during play.

3.4 Key Nodes and Training

The central theme of this paper is that a learner's ultimate proficiency depends upon its exposure to key nodes. Since human learners acquire knowledge that supports expertise from key nodes, direction to as many key nodes as possible should accelerate and strengthen the development of expertise. This opportunity occurs during training.

For our purposes, *training* is the learner's experience in the search space. In much of machine learning, a program is given a training set of experiences from which to learn. This set may be selected at random from a problem space, or it may be deliberately chosen, even sequenced, by an instructor. One way to evaluate a training experience would be to have an exhaustive list of key nodes with respect to a program's input knowledge and the search space, and then to check them off as they are encountered. This assumes that the space is extremely well understood by some person who can devise either a description for those key nodes or a procedure to enumerate them. For any challenging game, however, people cannot enumerate the key nodes, and the obvious exhaustive algorithm for their identification ("consider every possible set of nodes") is exponential in the number of nodes in the search space. A more pragmatic approach would be to indirectly assess the percentage of key nodes experienced during learning by testing the program's playing skill after learning.

A game learning program's opposition is, to some extent, its instructor, and is called here a *trainer*. The trainer has proclivities in its play which effectively shape the game learner's experience in the search space. The ideal trainer would be aware of the key nodes appropriate to a game learning program, its foreknowledge, and the game, that is, could compute the function f such that K = f(L, F, S). Such a trainer would play so that the learner would encounter every key node as quickly as possible. Once again, for a challenging game, this approach is intractable.

In any domain, how does a human or machine learner encounter key nodes? Here are some possibilities:

• Key nodes could arise by chance. Given the apparent relative scarcity of key nodes and the strength of expertise people develop, this seems unlikely. Among all 5478 possible tic-tac-toe nodes, only two (Figure 2(a) and its symmetric equivalent) are key nodes for Hoyle.

• Key nodes could be presented by a teacher as a sequence of training examples with explanations or solutions. A series of puzzle problems like Figure 2(a) for instruction would require that the teacher have a clear test for key nodes and a road map to reach them. Unfortunately, neither is available for difficult tasks; the only evidence that one has visited key nodes is in the improved quality of one's performance.

• Key nodes could be encountered in a carefully structured environment.

• The learner could seek key nodes out itself.

The last two possibilities, a structured environment and internal direction, have been explored with Hoyle, a program named for the eighteenth-century chess player Edmond Hoyle who wrote a book declaring official rules for popular games of the period. It was empirical work with Hoyle that initially led to a theory about key nodes.

4. Hoyle

Hoyle is a program that learns to play games during competition. Given the rules, Hoyle can play any two-person, perfect information, finite-board game correctly, and then learn to play it better, based upon its experience. Hoyle learns to play each game much the way the reader would to learn to play a strange game. You might not know anything at all about the game, but you do have a set of skills and expectations that would enable you to play according to the rules, and guide your efforts to learn to play well. Hoyle is based on a learning and problem-solving architecture called FORR, predicated upon multiple rationales for decision making (Epstein, 1994a). FORR supports *multiple learning strategies* (the ability to learn an item of knowledge more than one way, store alternative values, and formulate recommendations in a uniform manner based on alternative values), *multiple decision strategies* (the ability to make recommendations from more than one viewpoint), and *multiple knowledge application* (the ability to use an item of knowledge more than one way). A FORR-based program is expected to learn both by imitating expert behavior and by trying (and perhaps failing) to solve problems on its own. This section describes what Hoyle knows before it learns, what it is directed to learn, how it learns, and how Hoyle's learning impacts upon its behavior. Further details on Hoyle are available in (Epstein, 1992).

4.1 What Hoyle Knows before Learning

Hoyle has been carefully designed to facilitate learning about games, and learning about learning. Like many people, Hoyle comes equipped with a store of knowledge about game playing in general; it knows, for example, to try to make winning moves and to stop playing when one contestant has won. Unlike people, Hoyle can begin as a novice at any particular game as often as we like, and make explicit exactly what it knows and what it learns.

Hoyle simulates an expert game player who *in general* already knows how to play games (take turns, follow the rules), the good reasons for selecting a move (the Advisors described in Section 4.2), what to learn about a new game (the useful knowledge described in Section 4.3), and how to learn it (the learning algorithms described in Section 4.4). Given the rules of a new game (like those in the Appendix), Hoyle becomes an expert at the game by playing it. As it plays, Hoyle acquires *specific* useful knowledge about the new game, until it gradually makes better move choices based on that useful knowledge. When Hoyle learns to play a game, it is easy to distinguish between what the program knows in advance and what it learns, and to isolate knowledge about one game from knowledge about another.

4.2 How Hoyle Plays

Hoyle learns about a game by playing contests at it. A contest appears to Hoyle as a sequence of game states (nodes in the game tree) in which the mover makes a decision. A game state is described to the program as the location of the playing pieces on the board, which side is about to select a move, and whether Hoyle or its opposition is playing that side. A sample tic-tac-toe contest might begin with the three game states in Figure 3, where the empty positions are labeled with numbers.

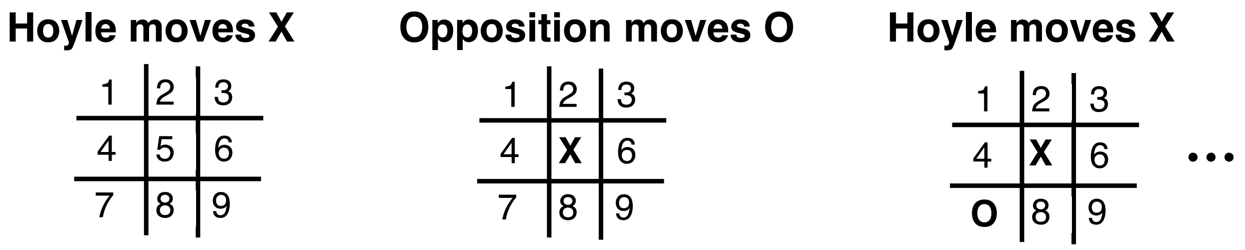

Figure 3. The beginning of a tic-tac-toe contest.

Hoyle is a *limitedly rational* program, that is, it implements methods known by its programmers to be reasonable but not necessarily perfect. The purportedly rational approach to game playing is to search down through the game tree from the current state to the leaves, and then reason back up to pick the best move from the current state. In tic-tac-toe it is relatively easy to calculate the best next move that way; in the enormous game tree for chess, it is usually impossible. Hoyle's Advisors are the core of its limited rationality, the commonsense strategies that help it select a move.

An *Advisor* is a resource-limited, game-independent, automated procedure that epitomizes a reasonable basis for selecting a move, like "it captures a piece" or "it wins the contest." When it is Hoyle's turn to move, the program consults its Advisors; they recommend for and advise against the current choices available. A full list of the Advisors appears in Table 2. The input to every Advisor is the same: the current game state, the current legal moves, and the current useful knowledge accrued for the game from experience. An Advisor's output is one or more *comments*, ordered triples naming the Advisor, a legal move, and an integer that indicates an opinion somewhere between strong aversion (0) and enthusiastic support (10). For example, in the rightmost state in Figure 3, input to each Advisor would be that state, the seven empty locations (1, 2, 3, 4, 6, 8, 9) on the grid that constitute legal moves, and whatever useful knowledge Hoyle has about tic-tac-toe. Comments the Advisors might make in this state include:

(Worried, move to 1, strength 8)

(Greedy, move to 1, strength 9)

(Patsy, move to 3, strength 2).

Table 2: Examples of Hoyle's Advisors

Advisor	Tier	Description	Useful knowledge	Learning Strategy
Wiser	1	Makes the correct move if the current state is remembered as a certain win.	Significant states	Deduction
Sadder	1	Resigns if the current state is remembered as a certain loss.	Significant states	Deduction
Victory	1	Makes the winning move from the current state if there is one.	None	—
Don't Lose	1	Eliminates any move that results in an immediate loss.	Significant states	Deduction
Panic	1	Blocks a winning move the non-mover would have if it were his turn now.	Significant states	Deduction
Shortsight	1	Advises for or against moves based on a two-ply lookahead.	Significant states	Deduction
Enough Rope	1	Avoids blocking a losing move the non-mover would have if it were his turn now.	None	—
Anthropomorph	2	Moves as a winning or drawing non-Hoyle expert did.	Expert moves	Abduction
Candide	2	Formulates and advances naive offensive plans.	None	—
Challenge	2	Moves to maximize its number of winning lines or minimize the non-mover's.	None	—
Coverage	2	Maximizes the mover's markers' influence on predrawn game board lines or minimizes the non-mover's.	None	—
Cyber	2	Moves as a winning or drawing Hoyle did.	Important contests	Abduction
Greedy	2	Moves to advance more than one winning line.	None	—
Leery	2	Avoids moves to a state from which a loss occurred, but where limited search proved no certain failure.	Play failure and proof failure	Abduction
Material	2	Moves to increase the number of its pieces or decrease those of the non-mover.	None	—
Freedom	2	Moves to maximize the number of its subsequent immediate moves or minimize those of the non-mover.	None	—
Not Again	2	Avoids moving as a losing Hoyle did.	Important contests	Abduction
Open	2	Recommends previously-observed expert openings.	Opening database Average contest length	Induction Induction
Patsy	2	Recreates visual patterns credited for positive outcomes in play; avoids those blamed for negative ones.	Visual patterns	Associative pattern classifier
Pitchfork	2	Advances offensive forks or destroys defensive ones.	Forks	EBL
Shortcut	2	Bisects the shortest paths between pairs of markers of the same contestant on predrawn lines.	None	—
Vulnerable	2	Reduces the non-mover's capture moves on two-ply lookahead.	None	—
Worried	2	Observes and destroys naive offensive plans of the non-mover.	None	—

Note. These procedures are general rationales that reference one or more items of useful knowledge, each supported by its own learning strategy.The algorithm for each Advisor is intended to capture the particular perspective it adopts, and the strength of a comment is determined differently by each one. Victory, for example, is fairly simple; it tests each legal move to find one that creates a winning state for the current mover. Pitchfork, on the other hand, is quite elaborate; it constructs a complex, insightful representation of the current state, and then relates it to a useful knowledge item called *forks* (Epstein, 1990). Each Advisor is implemented as a procedure with a time limit. To construct their comments, some Advisors are permitted to search exhaustively at most two moves ahead, that is, they look at all the possible states that could occur if Hoyle made every legal move and then the other contestant did the same. From the last state in Figure 3, for example, without symmetry there would be 7 ◊ 6 = 42 such possible states. Although Advisors are intended to be generally applicable, some Advisors may never comment for a particular game or may be irrelevant. For example, Material focuses on piece capture; it is irrelevant in games like tic-tac-toe where the number of pieces on the board monotonically increases with every move.

The role of the Advisors is to provide opinions upon which move selection is based. Not all 23 Advisors are equally reliable or equally important, so, as in the schematic of Figure 4, they are organized into two tiers. The seven Advisors in the first tier have a priority ranking based on commonsense knowledge, such as “try memory before computation.” They sequentially attempt to compute a decision based upon correct knowledge, shallow search, and simple inference, such as Victory’s “make a move that wins the contest immediately.” If no decision is forthcoming, then all 16 Advisors in the second tier collectively make their many, and often less reliable, comments based upon their narrow viewpoints, like Material’s “maximize the number of your playing pieces and minimize the number of your opponent’s.” The move with the most support is chosen, and ties are broken by random selection.

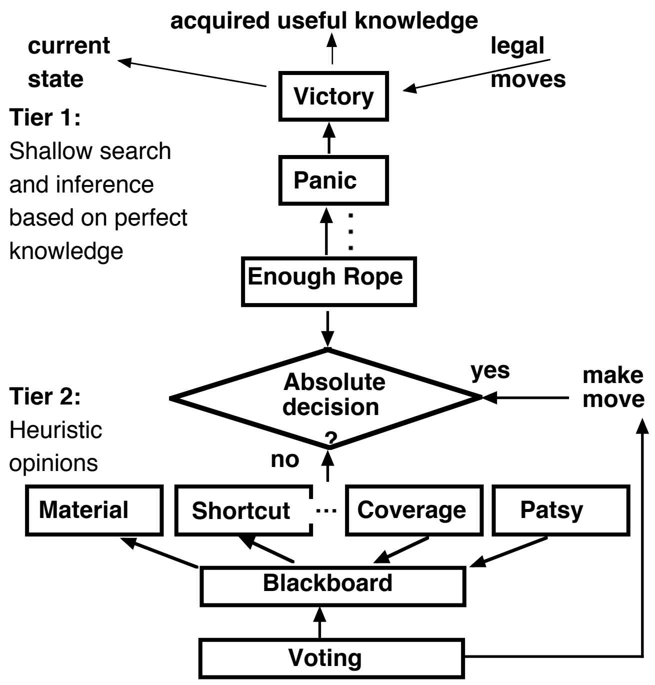

Figure 4. Hoyle makes decisions based upon comments from two tiers of Advisors.

4.3 What Hoyle Learns

What Hoyle learns is *useful knowledg*e, data about a game that is possibly relevant to its future play and probably correct. A useful knowledge item is a good question to ask about a new game; its value is an answer to that question. The questions are the same for every game, but their answers typically differ from one game to another. Thus useful knowledge in Hoyle is game-dependent data computed and stored in a game-independent manner. There is one learning procedure for every item of useful knowledge.

Openings are an example of useful knowledge. (Recall that an opening is a sequence of moves with which two competitors begin a contest.) The Advisor Open recommends moves that forward previously successful openings, and advises against moves that forward previously unsuccessful ones. Before the lose tic-tac-toe contest that began with Figure 2(b), Hoyle's useful knowledge slot for openings might have been empty. After the contest, one impact of learning might be that the lose tic-tac-toe useful knowledge slot for openings would now contain a representation for "open in the center, get a draw."

Another example of useful knowledge is dangerous states. The Advisor Leery recommends against, but does not forbid, moves that lead to dangerous states. For a few seconds after a non-draw contest, Hoyle searches exhaustively from the last unforced move of the loser, looking for a non-losing alternative. Because the allotted time is severely limited, such a search may prove inconclusive; if so, the state from which it began is learned as "dangerous." Before the tic-tac-toe contest that included Figure 2(a), Hoyle's useful knowledge slot for dangerous states might have been empty. Unless Hoyle has enough time after the contest to prove that a corner response is a certain loss, one impact of learning might be that the tic-tac-toe useful knowledge slot for dangerous states would now contain a representation for Figure 2(a) with an additional O in some corner and X as the mover.

Hoyle does not create an explicit representation of any game tree. Instead, Hoyle's useful knowledge is a compendium of ways to behave: a set of openings with some history about their success, a set of dangerous states with warnings to avoid them, and so on. Such information might be represented in many ways, for example, as explicitly formulated rules or as procedures with knowledge embedded in them. Because it is a FORR-based program, however, Hoyle represents this information as a set of game-specific knowledge caches (for example, lists or hash tables) plus a set of game-independent procedures (the Advisors) that exploit the caches. Each kind of useful knowledge is represented so that the Advisors that reference it can use it most efficiently.

4.4 How Hoyle Learns

Hoyle learns useful knowledge both after contests and after tournaments. Each item of useful knowledge is associated with at least one heuristic, game-independent learning procedure. Each learning algorithm has its own *trigger*, a condition whose truth initiates execution. Triggers can be tested after decisions, after contests, or after tournaments. The trigger for openings, for example, is Hoyle's failure to win a contest, and it is tested after every contest. Whenever a useful knowledge item's trigger is tested and found true, the learning algorithm for it is executed.

Thus, useful knowledge items are like perpetually recurring questions about experience, and their associated algorithms are procedures that attempt to construct answers.

The learning strategy varies from one procedure to the next, and may include explanation-based learning, induction, and deduction. For example, openings are learned by rote, just the way chess masters learn them. Hoyle simply records every opening, along with whether the contestant using it won, lost, or drew. The selection of a learning strategy for a particular item of useful knowledge is purely pragmatic, and often modeled on what we have observed in expert humans. Deduction amounts to proof in the game tree and is relatively expensive, that is, takes much time and space. It does, however, provide very strong support to the Advisors in the first tier, and is well worth some limited expenditure of resources. Induction bootstraps off a limited sampling and can be quite effective in certain situations. Although its results may be less trustworthy, they are often good enough. Abduction ("p achieves q and q happened so p must also have happened") is a flawed learning strategy that people often apply, with surprisingly good results. The "probably correct" nature of useful knowledge is directly attributable to the prevalence of non-deductive learning strategies. The learning algorithms are highly selective about what they retain, may generalize or abstract, and may choose to discard previously acquired knowledge.

4.5 How Learning Affects Hoyle's Behavior

Hoyle plays better when its Advisors have more complete and more correct knowledge to guide its decision making. Because many of its individual Advisors apply current useful knowledge to construct their comments, Hoyle learns from its experience to make better decisions from acquired useful knowledge.

Consider, for example, significant states. These are states in the game tree where, although the contest is not yet over, perfect play by one contestant will result in a win for that side no matter how well the other contestant plays. With experience an expert player should learn to recognize significant states and learn how to exploit them when they offer her side the advantage. An example of a *significant win state* (one that must result in a win for the mover) for

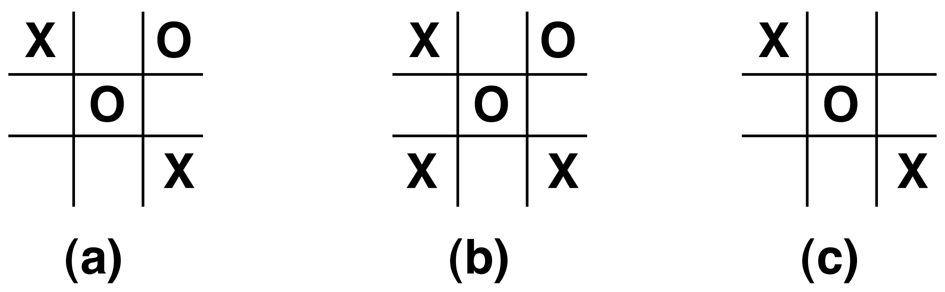

Figure 5. (a) A significant win state from tic-tac-toe, with X to move. This is not a leaf, but with perfect play X will win. (b) A significant loss state from tic-tac-toe, with O to move. O cannot prevent a perfect-playing X from a win. (c) A key node for Hoyle from tic-tac-toe, with O to move. This is not a significant state; with perfect play on both sides a draw will result.

tic-tac-toe appears in Figure 5(a) with X to move. An example of a *significant loss state* (one that must result in a loss for the mover) appears in Figure 5(b) with O to move. By definition, all the children of a significant loss state are themselves significant win states for the other contestant, and at least one of the children of a significant win state is a significant loss state for other contestant. Once a significant win state is learned, the Advisor Wiser recommends moves to it, and when Shortsight looks ahead two-ply it opposes choices that will in turn afford a move by the opposition to a significant win state of its own.

The relationship between key nodes and significant states is non-trivial. Consider, for example, a state S with a dozen children. After playing one or more contests including each of S's children, a learner might first deduce that each of S's children is a significant win state for the same contestant, and thereby deduce that S is a significant loss state for the other contestant. This does not mean that S and all its children are key nodes; perhaps visiting S alone would dissuade the learner from visiting it again. Thus a significant state is not necessarily a key node.

It may also happen that important learning occurs at a state that is not itself a significant state. Figure 5(c) is a replica of Figure 2(a), a key node for Hoyle at tic-tac-toe. The program will always make the wrong choice the first time it encounters this state. After the contest that includes Figure 5(c) or its symmetric equivalent, Hoyle learns that Figure 5(b) or its symmetric equivalent is a significant loss state for O. Thereafter, with two-ply lookahead, Hoyle will always refuse to play a corner in Figure 5(c). Clearly, significant states and key nodes are in some sense intertwined, and lie in each other's vicinity, but they are by no means equivalent.

5. Experiments with Hoyle: Visiting Key Nodes via External Direction

Without an explicit road map to key nodes, the most obvious way to expose a learner to them is to have the opposition play so that the learner must confront them. The data reported in this section is taken from extensive experiments in which Hoyle was required to learn to play three quite different draw games with small search spaces. Full details on that work are available in (Epstein, 1994b); here we cite data to demonstrate how difficult it is to reach key nodes.

In each experiment, Hoyle was expected to learn to play a game in competition against a hand-crafted, external program, called a *trainer*. This *learning tournament* went on until the program had learned to play well enough to draw 10 consecutive contests. Then learning was turned off and Hoyle played a twenty-contest *testing tournament* against each of four programmed *challengers*: a perfect contestant and routines that simulated a (slightly imperfect) expert, a novice, and a random contestant. Although the results were consistent for all three games, we recount here only those for lose tic-tac-toe, the most difficult of the three for the program to learn, and therefore the one that made the impact of external direction clearest. (Lose tic-tac-toe is difficult for Hoyle to learn because perfect play involves a concept it cannot easily represent -- symmetric movement reflected through the center -- and some fairly elaborate algorithms for achieving a win as O if the opening move is not the single correct one.)

There are a variety of competitors in this experiment: Hoyle, a perfect contestant, and some imperfect contestants. The perfect contestant is one that always makes the best possible move. If there is more than one such move, a perfect contestant chooses one at random to offer a broad variety of high-quality opposition. (A perfect contestant does not, however, set out to teach lessons about the nature of the game tree in any organized manner.) A single routine models all of the imperfect contestants. It accepts an error parameter that determines what percentage of the time to make a randomly chosen, legal, possibly perfect move, instead of a perfect one. For testing, we set this error parameter to simulate the expert, novice, and random challengers at 10%, 70%, and 100%, respectively. For training, we created a spectrum of *fallible trainers* with error parameters 10%, 20%,…, 100%.

In each of these experiments there is one learner (always Hoyle), one trainer, and the same four challengers: the perfect contestant and the expert, novice, and random challengers. The trainer is Hoyle's opposition during learning; the challengers are Hoyle's opposition during testing, after learning is turned off. The trainer varies from one experiment to the next; it may be the perfect contestant, a fallible trainer, or even Hoyle itself.

Table 3 highlights representative data; each value represents an average over 5 runs. We define *power* in this experiment to be the ability to exploit the other contestant's errors; it is measured here for a draw game by the percentage of contests won, and is reported against the expert, novice, and random challengers, in that order. (In a draw game, power against a perfect contestant must be zero.) We define *reliability* in this experiment to be the consistent achievement of an expert outcome; it is measured here for a draw game by the percentage of contests won or drawn, and is reported first against a perfect contestant, and then against the other challengers in the same order. In a draw game, ideal reliability is 100%. For lose tic-tac-toe, maximal power was computed in a 10,000-contest tournament between the perfect contestant and each of the other challengers. Maximal power is 16% against the expert, 66% against the novice, and 74% against the random challenger. Space is long-term memory allocation for useful knowledge; time is the number of contests the learning tournament ran until Hoyle was able to win or draw 10 consecutive contests.

Table 3: A Comparison of the Power and Reliability Achieved after Different Instruction in Lose Tic-tac-toe

	Learning with	**Power**	**Reliability**	**Space**[a]	**Time**[b]
1	Perfect contestant	27--65--**73**	**100--88--82--81**	89.2	35.6
2	10% fallible	16--70--61	57--91--84--78	135.6	45.6
3	20% fallible	15--54--68	60--96--86--79	224.2	68.8
4	30% fallible	31--59--68	57--78--78--82	137.6	40.6
5	40% fallible	24--65--78	58--89--81--86	325.6	95.2
6	50% fallible	28--61--74	49--71--84--82	306.8	87.6
7	60% fallible	35--61--78	52--72--81--84	273.8	77.0
8	70% fallible	29--69--77	41--63--83--82	301.6	85.6
9	80% fallible	41--72--71	45--68--80--85	154.8	42.8
10	90% fallible	32--60--76	42--72--78--88	209.2	62.0
11	100% fallible	41--64--75	57--66--80--87	185.4	57.8
12	Self-training	14--66--69	39--54--77--78	186.2	71.4
13	Lesson and practice	17--63--77	95--92--88--90	323.0	122.7
	Longer training				
14	Perfect contestant	12--59--80	100--93--80--88	101.0	49.3
15	Lesson and practice	18--63--**85**	**100--98--97--100**	299.0	127.7

Note. The last two columns detail the memory and experience requirements for such learning. The first 13 lines used a termination condition of 10 consecutive wins or draws; the last two lines used 20.

[a]Measured in items of useful knowledge.

[b]Measured in contests played.

A perfect contestant is an inadequate trainer because it provides no experience with key nodes beyond those that two perfect contestants would proffer to each other. The first line of Table 3 demonstrates that there are key nodes beyond those regularly visited during competition against a perfect contestant. Even though Hoyle learns to play perfectly reliably (100 as the first entry in column 3) against this trainer, its achievement is fragile: when confronted during testing with the imperfect moves of the other challengers it has difficulty with situations that arise from their errors. Although an expert should play better against a weaker contestant, Hoyle is *less* reliable against the other challengers than it is against the perfect contestant (88, 82, and 81% in column 3). Instruction with a perfect contestant shows the learner ideal play, but only in an extremely narrow context, as evidenced by the relatively small long-term memory (in column 4) allocated for useful knowledge. For example, a key node like Figure 5(c) may never arise in competition against a perfect contestant for which equally good moves are equally likely. Quite reasonably, such constrained experience is also brief; the learning time against the perfect contestant is significantly shorter than with other trainers.

Noise in the trainer does not lead the learner to more key nodes, that is, key nodes are not randomly distributed. A series of *fallible trainers* was tested, each a perfect contestant with a stochastic opportunity to err. The random move selection rates tested were multiples of 10% from 10% to 100%, inclusive. Lines 2 through 11 of Table 3 show representative data for the fallible trainers. Although some fallibility in the trainer provides Hoyle with expanded experience, too much fallibility makes the trainer so easy to defeat that Hoyle wins or draws 10 contests in a row without learning enough to do well in testing. This is why learning times peak midway through the last column of Table 3. Along with an increase in learning time and a randomized component go an increase in the number of nodes the program is likely to encounter, and an increased demand on long-term memory. If those additional nodes were equally likely to be key nodes, then the program should have learned to play better with more fallible instruction. In fact, with a highly fallible trainer Hoyle generally played worse, further evidence that key nodes are organized some way within the search space, a way that a perfect contestant somehow reflects. (Another factor in learning against a fallible trainer is imitation. One of Hoyle's second-tier Advisors imitates the other contestant as if it were a model of good play. When the other contestant makes a mistake, Hoyle can eventually unlearn it, that is, learn to prevent its replication, but that takes time and slows the development of expertise. Because imitation is one among many factors, a fallible trainer is not an insurmountable obstacle, but it certainly adds to the learner's confusion.)

Left on its own, a program is unlikely to encounter key nodes. When the trainer is eliminated, its lack of guidance to key nodes proves costly. In *self-training* Hoyle is expected to learn while playing against itself. Although self-training is often presumed to be a natural environment in which to improve expertise gradually, the data in line 12 of Table 3 indicate otherwise. With self-training, Hoyle was the least reliable against every challenger. Although Hoyle had achieved 10 consecutive draws during self-training, the nodes it chose to visit alone did not include enough of the knowledge learnable at key nodes that would later be required in an encounter against the challengers. Self-training is not particularly fast but it appears fairly repetitive. While self-training takes about twice as long as learning against a perfect contestant, it retains far less useful knowledge than other training that requires about the same number of contests.

A new training paradigm, called *lesson and practice training*, provides good guidance to key nodes. Lesson and practice training advocates interleaving relatively few learning contests against the best available expert (the *lesson*) with relatively many self-training contests (the *practice*). The application cited here gave Hoyle experience in cycles of 2 lessons followed by 7 practice contests. Lesson and practice training in line 13 of Table 3 shows some improvement over training against a perfect contestant in line 1; its increased reliability against the weaker challengers indicates a less fragile expert. Performance against the stronger challengers in line 13, however, is less satisfactory.

The problem, we suspected, was that training against a perfect contestant offers repeated opportunities to observe wisdom in action, while this particular version of lesson and practice training offers a lesson only 2/9 of the time. To determine if longer training would solve this problem by offering more exposure to high quality play, we reran lesson and practice training with a termination condition of 20 consecutive wins or draws instead of 10, and found an even more dramatic improvement, shown in line 15. Of course, this necessitated a suite of additional experiments to offer this potential advantage to the other trainers as well, and to test whether 50, or 75, or even 100 consecutive wins or draws was a better termination condition than 20. In most cases performance did not simply improve, or continue to improve, that is, there was no statistically significant improvement with later termination, or only a single instance of slightly improved performance that never approached line 1. The single exception, as reflected in line 15, was lesson and practice training with termination condition 20. (Line 14 is provided only for comparison.)

Lesson and practice training proved more reliable and powerful against all the challengers than most other kinds of instruction; it was never less reliable or less powerful at the 95% confidence level. Our explanation for this improved performance is that Hoyle was able to derive useful knowledge because it was forced by lesson and practice training to visit more key nodes during learning. Increasing the termination condition to 20 gives Hoyle (on average) one additional lesson, enough to help it simulate a robust expert.

Lesson and practice training is really an experiential style. It prescribes relatively few lessons (experience with an expert) followed by relatively extensive practice (exploration on one's own). This treats the lesson giver (in this case the perfect contestant) as a scarce resource, an attitude appropriate in many real-world situations. It also drives the learner's experience to places in the search space it would not experience during lessons. During practice, the learner will attempt to apply what it knows to a variety of related problems. As long as the learner's knowledge does not support perfect play, it is likely to make mistakes, particularly during practice, mistakes that take it to key nodes where it can learn to play better. We believe that practice works because the learner has good reasons (Hoyle's Advisors) and fairly reliable information (useful knowledge) on which to base its (perhaps misguided) decisions, and because it can learn from its mistakes, so that its experience refines its knowledge. Support for this theory appears in the next section.

6. Experiments with Hoyle: Visiting Key Nodes via Internal Direction

Hoyle is currently learning nine men's morris, an ancient African game played with 18 pieces on a 24-position board. Although it is a draw game, most people find the search space of 7,673,759,269 nodes quite challenging (Gasser, In press). Hoyle's contests against an external expert program average 60 moves in two stages: a *placing stage* with an average branching factor of 15.5, and a *sliding stage* with an average branching factor of 7.5. Only in a game this difficult has the impact of internal direction on Hoyle become evident.

Originally, Hoyle was not very good at nine men's morris; it lost every contest against its trainer, a hand-crafted, very strong, expert program. Recently, however, two new Advisors were added that capitalize on the visual cues provided by predrawn lines on the game board (Epstein, Gelfand, & Lesniak, In press). With these Advisors, the program initially had some ability to draw, and then began to learn to win. Table 4 shows the outcome of a 50-contest tournament with both the new Advisors in place. During the 50 contests in Table 4, Hoyle lost 24 times, drew 17 times, and won nine times. (Since nine men's morris is a draw game, the wins mean that the hand-crafted program made errors. We have located and are in the process of correcting some of them. There is, however, no program that plays this game perfectly. The proof that the game is a draw consists of a single, computer-generated path through the game tree that relies on a 2-gigabyte database generated by full retrograde analysis (Gasser, In press).)

Table 4: The Outcome of a 50-Contest Tournament at Nine Men's Morris between Hoyle and a Hand-crafted Expert Program

#	Outcome	#	Outcome	#	Outcome	#	Outcome	#	Outcome
1	draw	11	loss	21	loss	31	loss	41	draw
2	loss	12	loss	22	draw	32	loss	42	draw
3	draw	13	loss	23	loss	33	**win**	43	**win**
4	loss	14	draw	24	draw	34	draw	44	draw
5	loss	15	loss	25	loss	35	**win**	45	**win**
6	draw	16	draw	26	loss	36	loss	46	**win**
7	loss	17	draw	27	**win**	37	loss	47	**win**
8	draw	18	draw	28	loss	38	loss	48	**win**
9	loss	19	loss	29	loss	39	**win**	49	loss
10	draw	20	loss	30	draw	40	loss	50	draw

Note. "**win**" indicates that Hoyle defeated the trainer; "loss" indicates that Hoyle lost to the trainer.

The first win was not until the 27th contest, and five of the wins were in the last eight contests, suggesting that Hoyle was learning to play better as it acquired more experience. (The likelihood that when 10 wins are distributed at random among 50 contests, none would appear before the 27th contest is .02%, and that five would appear in the last eight is 0.4%.) Comparison against what we believe to be a perfect contestant for the placing stage indicates that the program made no identifiable errors in the placing stage of any of the last 10 contests.

There are several possible explanations for this clear improvement:

• *New application of useful knowledge:* One possibility is that the new Advisors were associated with a new category of useful knowledge, but these two do not reference any useful knowledge at all, and so could not benefit from learning.

• *Unusually accurate new Advisors:* Another possibility is that the new Advisors were so clever that they quickly dominated the decision making process. This is not the case either; Hoyle gathers statistics on the performance of its Advisors and, although the new ones were active, they by no means overwhelmed the others. Although there were now some early draws, the ability to win was not immediate, as it should have been if these Advisors "knew" the right moves from the start; the program's improvement was clearly gradual.

• *Direction without learning:* Yet another possibility is that learning was already drawing the program to places where all Hoyle needed was some good advice from these new Advisors to play well. If that were so, then the useful knowledge store should be roughly the same size without the new Advisors as with them. This is not the case; the useful knowledge store increased with the new Advisors.

• *Visiting the key nodes:* The final possible explanation, and the one advocated here, is that *the program performed better because of the new nodes it chose to visit during learning*. With different useful knowledge as input, the same Advisors in the same states are likely to make different comments, and based upon those different comments Hoyle is likely to select a different move. The new Advisors drew Hoyle to places in the search space where its learning algorithms extracted additional, more powerful, useful knowledge upon which the other

Advisors were able to capitalize. Hoyle definitely plays smarter with these new Advisors, smart play directs it to the key nodes, and Hoyle learns enough there to make even better decisions than it did without the new Advisors, or with the new Advisors but without learning.

7. Discussion

The results described here support hypotheses about the nature of a learner's task and teaching toward it, and offer a perspective on learning programs.

7.1 Exploiting Key Nodes: The Learner's Task

This paper addresses the ability of a human learner to bootstrap from limited experience in a very large problem space to robust expertise throughout it. Section 5 offers evidence that some problem instances (key nodes) can provide important developmental experience, and that these instances are not randomly distributed in the search space. Section 6 offers evidence that knowing *how* to find those key nodes is extremely important.

Indeed, a sudden, rather than a gradual, improvement in performance during learning may be explainable as the crystallization of developing internal direction, as if the learner has finalized some good decision-making technique. The impact of new Advisors on Hoyle's ability to learn nine men's morris supports this hypothesis. Current research is successfully exploring ways to have the program learn new Advisors on its own that appear at this writing to have a similarly beneficial effect (Epstein, et al., In press). Another example of sudden improvement appears in TD-gammon, the neural net program that learns to play backgammon as well as the best human experts (Tesauro, 1992). Work on TD-gammon supports the principles of well-guided search. The program originally learned in about a month, playing 200,000 contests against itself. Contests averaged 60 moves, so that the program encountered no more than 2◊ 10^8 nodes, a tiny fraction of its search space. During the first week, however, the program played "long, looping contests that seemed to go nowhere" (Tesauro, 1991, personal communication). Then the program improved rapidly, suggesting that first the network learned to find key nodes, and *then* to make good decisions. A less accomplished precursor, Neurogammon, had a trainer, but repeatedly experienced the same 400 contests, without any ability to encounter additional key nodes. When TD-gammon learned against Neurogammon instead of against itself, it was able to progress away from random moves much more quickly (Tesauro, 1991).

The ability of people like grandmasters to function at a high level when they have only experienced a tiny fraction of a search space suggests, as noted in Section 3, that key nodes are not evenly distributed. A person studying chess will often deliberately explore multiple alternatives from a state that arose during play, and find that such search clarifies some difficulty by driving the examination back to the problem's origin in earlier play or by driving it forward to some resolution. If one envisions those small searches as paths in a restricted neighborhood or *cluster*, we hypothesize that such clusters are particularly rich in key nodes, and that human game learners deliberately traverse them because such clusters offer a particularly effective learning experience. The evidence we offer for this theory is Hoyle's improved performance at the same games under lesson and practice training. Lesson and practice training is driven by Hoyle's Advisors and its game-specific useful knowledge. In the laboratory we have regularly observed Hoyle using practice contests to explore in the vicinity of nodes it had experienced during lessons. Hoyle's Advisors rely upon useful knowledge while they encourage cluster exploration by devices like the repetition of learned openings (Open) and the imitation of expert (Anthropomorph) and successful (Cyber) play.

7.2 Well-Guided Search: The Teacher's Task

There are clear lessons in this work for those who teach learners like Hoyle and, to the extent that Hoyle is human-like, for those who teach people. In a difficult problem space, there is far too much to remember by rote, and not all experience is equally worth retaining. Given that random experience is unlikely to take the learner to enough key nodes in an otherwise intractable problem space, the role of a teacher is to provide guidance there, and this supervision is non-trivial.

• A lack of instructional breadth, like training with a perfect contestant, may overlook nodes that the nascent expert should visit for robustness. A correct model of behavior is insufficient because it is too rigid in its paths through the search space.

• A lack of instructional reliability, like training with fallible contestants, may fragment experience and distract the learner from causal associations. In game playing, for example, the problem state representation is always complete, in the sense that no information is excluded, that is, the identity of the mover and location of every playing piece is specified. Thus the randomness introduced by a fallible trainer can vary *any* aspect of the learning situation, whether or not it is relevant to the problem of learning to play well. Whether or not the learner can manage to extract some useful knowledge from such random variation depends in part upon the distribution of key nodes in the search space. If key nodes are relatively sparse in the space, the randomness of a fallible trainer is unlikely to drive the learner to a key node.

• Left to its own devices, a learner is unlikely to encounter key nodes either. Self-training is likely to combine a lack of expertise with a lack of inherent variation. Instead, the learner needs both to be led *and* to explore the problem space alone, preferably by perturbing its behavior based upon knowledge. High quality instruction requires variation that addresses the vicinity of the key nodes; that is what lesson and practice training appears to do. The lessons direct the learner to the nodes an expert would be likely to visit, while the practice permits the learner to explore variations on that guidance. (Although lesson and practice training was modeled on skill development in chess players, the author recognizes its similarities to classroom instruction and homework.) Unlike fallible training, the non-random variation of lesson and practice training, motivated in Hoyle by Advisors and useful knowledge, apparently leads to key nodes and to more successful learning.

7.3 Making it Work: The Programmer's Task

There are also more general lessons here for the development of expert programs that learn. An evaluation function in a learning program should be constructed not only to make good decisions but also to route exploration to guide learning. The typical, competitively successful game-playing program has incorporated only some of the work from cognitive science: an extensive *opening book* (early move sequences favored by human experts) and a store of *endgame knowledge* that describes the best way to finish a contest. It is in the middlegame where such a program makes most of its mistakes, when it selects a move with a heuristic approximation of exhaustive search. The program relies upon a prespecified, game-dependent set of *features*, properties of a state like “king in check” or “control of the center.” The program’s *evaluation function* is a game-dependent metric for the goodness of a state, computed as some combination of the values of the feature set at that state. Middlegame decisions are dependent on the accuracy of a feature-based evaluation function and, to some extent, on the search depth. This paper shows that, unless those features direct the program’s attention to key nodes and unless the program can learn there, middlegame play will be both inefficient and inaccurate.

A careful distinction between the expert’s successful behavior and the learner’s is constructive. For most challenging games, one must presume an imperfect trainer. If a program takes an imperfect trainer as a model of expert behavior, however, it will eventually learn something that is incorrect. One reason lesson and practice training succeeds is probably that Hoyle distinguishes carefully between its own moves and those of its trainer. A trainer’s move is more highly regarded, and trainer error is less likely to be suspected. Thus nodes are in some sense labeled by the reason that they were visited, and treated accordingly. If the trainer makes errors unidentified as such, the program may choose to imitate them (with the Advisor Anthropomorph) and thereby slow (but not disable) its own development of expertise. The learning program is expected to make errors, and tolerates them well, but is expected to make fewer as it has more experience.

A theory of well-guided search is applicable to domains beyond game playing. Recent work in cognitive science indicates that studies of chess experts are representative of expertise in other fields as well. Regardless of the domain, experts are distinguished from novices in several ways: they rely heavily on mental models and special-purpose knowledge representations, they have larger long-term memories and expert procedural techniques, and they have extensive, readily accessible knowledge (Chase, et al., 1973; Ericsson & Smith, 1991; Ericsson & Staszewski, 1989). Their search, however, is distinguished by its limited focus, not its breadth or speed (Charness, 1991). This information suggests that key nodes are present in other domains as well.

Like these experts, Hoyle has special-purpose knowledge representations (half a dozen ways to represent the board, for example), large long-term memory (dynamically allocated caches of thus far unlimited size), and expert procedural techniques (the Advisors). To date, however, Hoyle's search is more severely restricted and less focused, and its memory organization less conducive to fast retrieval than human experts'.

Finally, the work described here highlights the dangers inherent in learning with a *reactive* program, a goal-free program that simply senses its environment and responds to it (Brooks, 1991). A reactive program is in some sense the victim of its environment; it is expected only to experience, generalize and correct, and keep reacting. It has no control over the nature of its next experience, and can take no initiative. Since the environment *is* its trainer, a reactive learning program can only succeed to the extent that its environment shapes its experience appropriately. Thus a reactive learner must rely on its environment to deliver it to key nodes in far greater proportion than they appear in its search space.

7.4 Additional Issues

There are many interesting issues associated with the theory of well-guided search as proposed here. The identification of key nodes is non-trivial; although Hoyle has some successful, domain-dependent detectors, "important" or "interesting" remains an elusive property. The distribution of key nodes, in particular their tendency to cluster, may vary with the domain. Perhaps combinations of key nodes, rather than individual ones, are what drives learning. (Indeed, we have observed in our laboratory individual runs where fortuitously-encountered sequences of key nodes in early training produced exceptionally rapid learning.) Different learning strategies may demand different key nodes. Explicit labeling of key nodes may even suggest a new class of learning strategies that rely upon their significance as exemplars or counterexamples, or require clusters of key nodes as input. All of these are topics for future work.

A program needs to learn knowledge that will correct its expectations about each state the next time. Rather than begin each contest from the initial state, a game-learning program could begin a practice session from such a key node. This would still not be exhaustive search, but a series of carefully organized “what if’s” that addressed problematic positions. A surprised learner could, quite appropriately, query its trainer or begin a line of inquiry along with or in competition against its trainer. A game-learning program might deliberately construct a state (or a set of states) from which to begin a practice session. For example, if a program encounters a new opening and loses to it, why should it have to wait until it encounters that opening again to seek an appropriate defense? The program could instead undertake a deliberate investigation of the opening, alternately playing each side, preferably against the same expert who introduced it. Here the key node (or key nodes) may expose important strategic strengths or weaknesses.

Hoyle already saves contests that it considers significant because they violate its expectations. Key nodes arise in the context of good teaching cases. Thus a natural extension of a skilled learner to a skilled teacher would be to focus on these paradigmatic experiences. Where the program has learned, so may its students, particularly if the instructor and the students have similar learning algorithms.

8. Related Work

8.1 Computers and Game Learning

Often, the computer simulation of expertise is simply a representation by the programmer of a formula like the one to direct the driver, that is, the program is *created* as an expert with knowledge about the entire space. To impart that expertise, the programmer must first calculate it or extract it from a human expert. Most expert game-playing programs are like this. The best of them usually achieve their prowess with an opening book, a *search engine* that drives the program to explore and estimate the winning potential of millions of possible future situations per second, and, in some cases, a store of endgame knowledge (Anantharaman, et al., 1990; Berliner, et al., 1989; DeJong & Schultz, 1988; Schaeffer, et al., 1992; Schultz & De Jong, 1988).

For complex problem areas like a large unmapped country or a difficult game, however, exhaustive representation of knowledge about a space may be infeasible: people may not have the knowledge, or it would require too much time and computer memory to calculate and store it.

A more realistic approach is to design a program that *learns* to become an expert, like a driving program whose route selection formula evolves with its experience. Among the few game-playing programs that *learn* to play expertly, most are limited to a single game and a single learning method. The outstanding success among them is TD-gammon which learns weights for a neural network to select the best next move(Tesauro, 1992). This technique has met with disappointing results in other games, however (Boyan, 1992). Most game-learning programs also use a single learning method. Although a person might learn openings by rote and learn the value of almost-completed contest states by deduction, game-learning programs generally use only a neural net or only temporal difference learning or only explanation-based learning.

Aside from TD-gammon, game-learning programs tend to be overwhelmed by the highly detailed, possibly inaccurate knowledge they acquire. As a result, their performance is either unacceptably slow or not at the level of a human expert (Fawcett & Utgoff, 1991; Freed, 1991; Levinson & Snyder, 1991; Morales, 1991; Wilkins, 1980). Consider, for example, T2, which learned 45 predicate calculus expressions for tic-tac-toe with 52 exception clauses after 800 contests (Yee, Saxena, Utgoff, & Barto, 1990). There is no natural organization for this knowledge, and binding during search is costly. Clauses and exceptions may subsume each other and are oriented toward optimal (shortest contest length) play. Although these deduced truths support correct play, they afford too much information to provide real-time decision-making guidance. Such programs are acquiring exhaustive knowledge of the problem space, not just knowledge about how to play expertly there.

8.2 Driving Discovery

The distinctive property of *discovery learning* is that the learner is expected to formulate and direct its own tasks, rather than to follow the examples or the tasks set by an instructor. Discovery learning is notoriously difficult because the program must focus its own attention. Early work on mathematical discovery, for example, was found to be unintentionally biased by LISP, the language in which the concept definitions were formulated (Lenat, 1976; Lenat & Brown, 1984; Ritchie & Hanna, 1984). Focus of attention requires meta-knowledge: a measure of self-awareness and a metric for *interestingness*. A discovery program must therefore either begin with a bias as to what is interesting or must learn it.

If interestingness were domain independent, or if the program were restricted to a set of related domains, interestingness could be defined in advance. Three fundamental elements of the definition are *surprise* (i.e., expectation failure), *curiosity* (i.e., inconclusive knowledge), and *ignorance* (i.e., computation failure). Each of these is an important indication of inadequate knowledge and often cited by people as the reason they formulate a task.

Surprise occurs when a learner expects something different from what actually occurs, for example, a robot believes it has picked up a wrench and then determines that the wrench is still on the table. Surprising states are a kind of key node. Indeed, the position in Figure 5(c) is interesting *because* it violates the expectation that X will win. Langley's BACON program determined its explorations by surprise when data did not fit curves as it was expected to do (Langley, Simon, Bradshaw, & Zytkow, 1987).

Curiosity is spurred by inconclusive data. Pell has considered incomplete experiential data about the efficacy of a move in Go on a 9 ∞ 9 board (Pell, 1991). His program gathered statistics on the wins and losses associated with individual moves. It pursued moves that had a true mean of winning most likely to be better than a *fickleness* parameter. This amounts to a forced exploration of situations which are not clearly categorized by the feature set and the program's playing experience. Against an external, hand-coded expert, Pell's program with fickleness .5 performed better than a program that simply chose a move based on the historical data. In the context of key nodes, the primary difficulty with this approach is that it is directed toward move preference, rather than toward knowledge acquisition for expert behavior. Thus the learning it results in is reactive (what to do) and not particularly transparent.

Moore has a program for a learning control system that attempts to concentrate experience in portions of the control space relevant to the current task (Moore, 1991). His program introduces some randomness into an otherwise carefully learned controller. He shows that random perturbations of a decision are better than random decisions, and that predictive analysis of random perturbation is even more effective. This last class of methods attempts to harness some regularity in the space to identify good decisions. The tacit assumption in his approach is that the value of an action is to some extent continuous in the space. Although that may be true for direction in his toy car domain, it is not necessarily true in many other domains, including game playing.

Ignorance was the impetus to plan to learn, in the context of biological research, for Hunter's IVY (Hunter, 1989). This opportunistic planner generated knowledge goals and saved them until appropriate knowledge arrived in the system. An IVY-inspired example of ignorance-driven search would be to seek a situation in which one decision-making principle (Advisor) would be inapplicable or always take precedent over another one. Rather than attempt to construct such states, the program could save their partial descriptions and alert itself for analysis when they arose.

The brevity of this survey of related work shows that driving discovery is a little-addressed task, although its principles (surprise, curiosity, and ignorance) are now clearly identified. This successes of some of the work cited here bode well for giving a learning program (and perhaps a learning person) more control of its experience.

8.3 Key Nodes, Cases, and Exemplars

Two other ways to address the acquisition of expertise in a large search space from limited experience in it are case-based reasoning and exemplars. *Case-based reasoning* (*CBR*) attempts to draw an analogy between a *case* (an old, previously-solved and retained problem, such as a state from which a move must be chosen) and a new, unsolved one. If an analogy is found, the case-based reasoner tries to adapt the old solution to serve as a solution to the new problem. (See, for example, Ram, 1993; Veloso & Carbonell, 1993.) Cases are abstracted and then indexed by relevant features to speed retrieval. Given the right cases in a domain, and effective algorithms to match cases and to adapt solutions, a case-based reasoner should function expertly in a domain where it has limited experience.

There are several differences between case-based reasoning and the theory of key nodes as proposed here. Usually every experienced problem distinct after abstraction becomes a case, and is therefore retained. Retained cases serve much like Hoyle's useful knowledge; they are possibly relevant to future performance, and solutions constructed from them are probably correct. In ordinary CBR, however, all cases are equally important; the strength of their role in future problem solving does not distinguish them. Key nodes, on the other hand, are not necessarily retained, but are distinguished precisely because they make a greater than average contribution.

An *exemplar* is a case that serves as prototype for some members of its class (Porter, Bareiss, & Holte, 1990). Like a key node, an exemplar is distinguished from other experiences in the search space. An exemplar can provide so much information that other members of its class need not be consulted at all; a key node can lead to the acquisition of such powerful information that other nodes need not be visited at all. Unlike a key node, however, an exemplar is always retained and used as a primary source for the construction of a solution, while useful knowledge acquired at key nodes is in no way distinguished as special. Knowledge from key nodes can therefore be intermingled and applied in a variety of unanticipated ways, whereas the combination of exemplars is more complex.

Unlike key nodes, it is easy to show that cases and exemplars are used by people to reason and to teach. CBR is, however, generally intended as a response to experience in the problem space, rather than a constructor of its own new learning experiences. Thus guidance to new cases is not an issue, as it is for key nodes; one simply relies upon the trainer to present the relevant cases, or hopes for the best. The role of cases in learning is also different; they offer paradigms for solution construction, and are expected to provide a general-purpose plan. Key nodes, in contrast, are not expected to be near-answers, only worthwhile learning experiences.

A domain like game playing is, in any case, a poor candidate for CBR. Game playing lacks a powerful general feature language for indexing and must contend with a second, uncooperative and unpredictable contestant. In addition, game trees have two structural differences from the typical case-based reasoning domains: discontinuity and goal irregularities. Game trees lack continuity; often even the relocation of a single playing piece to an immediately adjacent position destroys any deep similarity between a state and its perturbation. In addition, although "… with all case-based systems, the assumption is that the domain itself is regular with regard to the goals that will tend to be conjoined," there is no guarantee of that in game-playing (Hammond, Converse, Marks, & Seifert, 1993, p.107). The goal in game-playing is to achieve the best possible outcome permitted by the structure of the game tree. Beyond that, there are many game-dependent subgoals, like queening a pawn in chess or making a mill in nine men's morris. A problem state might be generalized as an interaction among subgoals, if they could all be identified, noticed, and remembered. The current state of the art in game-playing programs, in this author's estimation, focuses upon overly-specific generalizations that overlook the themes that underlie play.

9. Conclusions

If the knowledge intrinsic to expert performance can only be extracted after all, or even most, of the nodes in an intractably large search space are visited, then prospects for a person to learn expertise there would be dim. *The existence of human experts in spaces intractable for them argues that most of the important knowledge is located at a limited number of key nodes*. Thus, visiting the key nodes should become a priority during learning in a very large space, regardless of the learning strategy used after arrival there.

Hoyle is predicated on the idea that general expertise in a broad domain can be efficiently instantiated for a subdomain to develop specific expertise there, as when general game-playing knowledge is applied to a particular game. For Hoyle, this is discovery learning, triggered by its experience during play. Although Hoyle exploits non-experiential knowledge (like the rules, the Advisors, and various knowledge representations), the program will learn nothing unless it plays. For human experts, too, experience is the catalyst for learning. Thus the focus on key nodes is appropriate.

Deliberate direction to key nodes is well-guided search. Given the goal of learning to perform expertly in a large space, a theory for well-guided search begins with the following principles:

• There are key nodes where important knowledge may be extracted by the learner.

• Key nodes appear in clusters.

• Guidance to those key nodes is both appropriate and necessary.

• Internal direction to key nodes is available from high-quality decisions made by the learner.

• External direction to key nodes can be managed by the trainer.

• A mixture of external direction to key nodes and exploration in their vicinity is a productive way to exploit the clusters of key nodes in a large space and to compensate for trainer error.

References

Allard, F., Graham, S. & Paarsalu, M. E. (1980). Perception in sport: Basketball. *Journal of Sport Psychology*, *2*, 14-21.

Anantharaman, T., Campbell, M. S. & Hsu, F.-h. (1990). Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, *43*(1), 99-110.

Berlekamp, E. R., Conway, J. H. & Guy, R. K. (1982). *Winning ways for your mathematical plays* . London: Academic Press.

Berliner, H. & Ebeling, C. (1989). Pattern Knowledge and Search: The SUPREM architecture. *Artificial Intelligence*, *38*(2), 161-198.

Binet, A. (1894). *Psychologie des grands calculateurs et joueurs d'échecs* . Paris: Hachette.

Boyan, J. A. (1992). *Modular neural networks for learning context-dependent game strategies*. Master's thesis, University of Cambridge.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, *47*(1-3), 139-160.

Charness, N. (1981). Search in chess: Age and skill differences. *Journal of Experimental Psychology: Human Perception and Performance*, *7*, 467-476.

Yee, R. C., Saxena, S., Utgoff, P. E. & Barto, A. G. (1990). Explaining temporal differences to create useful concepts for evaluating states. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 882-888). Boston, MA: AAAI Press.

Appendix

Game-Dependent Knowledge

HOYLE’s game-dependent prior knowledge is kept to a minimum. This section provides the rules for two games, tic-tac-toe and nine men’s morris, first as a human might define them and then as Hoyle is given them, in its *game frame*. This is the only kind of game-dependent information Hoyle is given before it learns to play. Each game frame consists of nine variables and eight functions. (There are a few additional descriptors, such as the number of boards that are displayed on the screen during play before it is scrolled. They support implementation but contain no relevant playing knowledge)

The functions in the game frame appear to Hoyle as “black boxes,” that is, Hoyle passes arguments to them and receives answers but cannot inspect them. For example, there is no way to determine if captures can happen in a game except by playing it. Directions for the user are only displayed on the screen to inform a human contestant. The move input reader communicates with a human contestant at the keyboard. The move filter tests that an input or calculated move obeys the rules. The display function draws the current state on the screen. The move effector applies a move to a state to produce the next state, that is, it makes a move. The legal move generator calculates the list of all rule-abiding moves from a given state. The endp, winp, and lossp predicates compute whether a given state is the last in a contest, and, if so, whether it is a win or a loss. The visualize function transforms a list-like board into an array representation; the devisualize function reverses that transformation. Unless the display graphics are elaborate, the functions referenced in the game frame typically require only about 70 lines of code in all. The code is object-oriented and contains no game-specific features or evaluation function.

A. Tic-tac-toe and Lose Tic-tac-toe

Tic-tac-toe is played on a 3 ¥ 3 grid. The contestant that moves first has five X’s; the other contestant has four O’s. Initially the board is empty. A turn consists of placing one of your playing pieces in any empty square. The first one to place three owned playing pieces in a row, vertically, horizontally, or diagonally, wins. There are eight such winning lines. Play ends in a draw when there are no more empty squares.

In the game definition for tic-tac-toe in Table A1, the squares in the grid are numbered from left to right, one row at a time, beginning with 1. The primary internal representation of every two-dimensional game board is a list. (Note that there is therefore no explicit representation of corner, center, or edge, although one may be computed.) Different Advisors use different representations, but the list-like one predominates. The only changes required to transform Hoyle's tic-tac-toe rules into lose tic-tac-toe rules would be to reword the directions for a human contestant and to interchange the way the winner and loser are computed.

Table A1: Hoyle's Rules for Tic-tac-toe

Name: *tic-tac-toe*

Token for Player: *X*

Token for Opponent: *O*

Initial board: (*NIL NIL NIL NIL NIL NIL NIL NIL NIL*)

Adjacency graph: *NIL*

Visible predrawn straight lines: *NIL*

If square grid, dimension: *3*

Piece may change location once played: *no*

Winning lines: ((*1 2 3*) (*4 5 6*) (*7 8 9*) (*1 4 7*) (*2 5 8*) (*3 6 9*) (*1 5 9*) (*3 5 7*))

Directions for the user: *directions-tic-tac-toe*

Move input reader: *reader-tic-tac-toe*

Move filter: *legalp-tic-tac-toe*

Display function for current state: *display-tic-tac-toe*

Move effector: *effector-tic-tac-toe*

Exhaustive legal move generator: *generator-tic-tac-toe*

Predicate to detect end of contest: *endp-tic-tac-toe*

Predicate to calculate winner: *winp-tic-tac-toe*

Predicate to calculate loser: *lossp-tic-tac-toe*

Visualize: *visualize-tic-tac-toe*

Devisualize: *devisualize-tic-tac-toe*

B. Nine Men's Morris

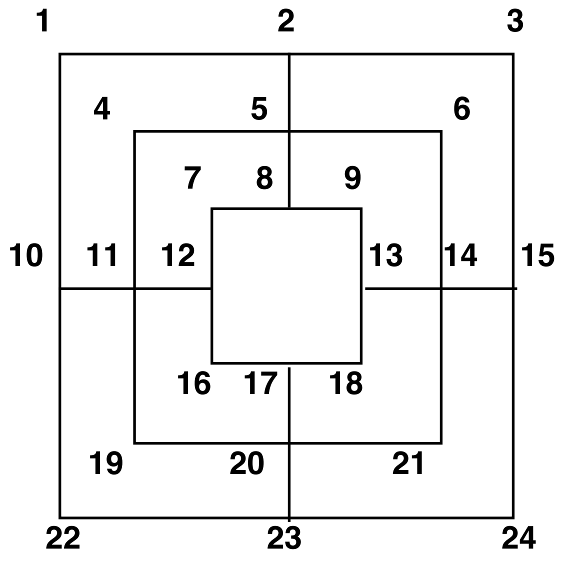

Figure A1. The game board for nine men's morris.

Nine men's morris is played on the game board in Figure A1. The contestant that moves first has nine black playing pieces; the other contestant has nine white ones. A playing piece may only rest on the intersection of two or more lines; there are 24 such positions. The game has two consecutive stages, a placing stage and a sliding stage. In the *placing stage*, the initial board is empty, and a turn consists of placing one's previously-unused playing piece on an empty position. Once all 18 playing pieces are on the board, the *sliding stage* begins, and a turn consists of sliding one's playing piece along a line to the next empty position. No playing piece may jump over another or be lifted from the board during a slide. Three of the same color playing pieces in a straight line along any side of any square (e.g., 1-2-3 or 7-12-16) is called a *mill*. Each time a contestant achieves a mill, she immediately removes a playing piece of the opposite color that is not in a mill. If all markers of the color to be removed are in mills, any such playing piece may be removed. Removed playing pieces never return to the board. The first one reduced to two playing pieces or unable to slide loses.

Hoyle's game definition for nine men's morris appears in Table A2. Note that there is no explicit representation of mill, nor of square, corner, center, or edge.

Table A2: Hoyle's Rules for Nine Men's Morris

Name: *nine-mens-morris*

Token for Player: *B*

Token for Opponent: *W*

Initial board: (*NIL NIL … NIL*)

Adjacency graph: ((*1 2*) (*1 10*) (*2 1*) (*2 3*) (*2 5*)… (*24 15*) (*24 23*))

Visible predrawn straight lines: ((*1 2*) (*1 10*) (*2 1*) (*2 3*) (*2 5*)… (*24 15*) (*24 23*))

If square grid, dimension: *NIL*

Piece may change location once played: *yes*

Winning lines: *NIL*

Directions for the user: *directions-nine-mens-morris*

Move input reader: *reader-nine-mens-morris*

Move filter: *legalp-nine-mens-morris*

Display function for current state: *display-nine-mens-morris*

Move effector: *effector-nine-mens-morris*

Exhaustive legal move generator: *generator-nine-mens-morris*

Predicate to detect end of contest: *endp-nine-mens-morris*

Predicate to calculate winner: *winp-nine-mens-morris*

Predicate to calculate loser: *lossp-nine-mens-morris*

Visualize: *visualize-nine-mens-morris*

Devisualize: *devisualize-nine-mens-morris*

Author Note

The development of Hoyle was supported in part by the National Science Foundation, #9001936. This work has benefited from conversations with Anders Ericsson, Jack Gelfand, Tom Mitchell, and Stan Matwin. The author takes full responsibility for any remaining obscurities. Pascal Abadie and Joanna Lesniak provided expert programming support.

Correspondence concerning this article should be addressed to Susan L. Epstein, Department of Computer Science, Hunter College, 695 Park Avenue, New York, New York, 10021. Electronic mail may be sent via Internet to sehhc@cunyvm.cuny.edu.