INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Learning and Applying Temporal Patterns through Experience

by

Esther Lock

A dissertation submitted to the Graduate Faculty In Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2003

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of the Doctor of Philosophy.

June 23, 2003
Date

Susan L Epstein
Chair of Examining Committee

July 3, 03
Date

Executive Officer

Dr. Jack Gelfand

Dr. Keith Harrow

Dr. Virginia Teller

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

Learning and Applying Temporal Patterns through Experience

By

Esther Lock

Adviser: Professor Susan Epstein

The thesis of this work is that the way patterns are formed over time can be learned during experience in a domain, and subsequently be used to improve decision making in that domain. Learning from temporal patterns can aid in the transition towards expertise, because it can increase the speed of decision making and improve the quality of the decisions made. While many researchers have explored ways to learn the significant patterns in a domain, the idea of focusing on the order with which these patterns are formed is novel.

This thesis investigates three learning methods that acquire and apply temporal patterns for game playing (i.e., sequences of game playing actions). *TP-Rote* (Temporal Patterns by Rote) is a rote-learning, caching scheme that hones in on frequently-used segments of a search space and memorizes them to reuse them later. Experiments with TP-Rote indicate a significant speedup in play, simulating the gradual shift to "play without thought" seen in human game players at various points in a game. The second method, *TP-Context* (Temporal Patterns through Context), generalizes states, retaining only the *context* (the motivating description) of a sequence. Such a generalization may cover many states and thereby expand the applicability of a temporal pattern. This is especially important in large state spaces, where identical states may not often recur. TP-Context exploits the knowledge inherent in the sequences actually experienced in game-

playing contests, to discover context. A learned context is associated with a sequence that TP-Context suggests as a course of action whenever the context is found on a given state. Experimental results show that TP-Context can learn to play two games, lose tic-tac-toe and five mens morris, based upon this sequence knowledge. The third method, *TP-Sitact* (Temporal Patterns through Context using a situation-action representation), is an extension of TP-Context. Although based on sequence knowledge, TP-Sitact suggests individual actions rather than sequences. TP-Sitact significantly improves TP-Context's game playing prowess. With these three methods, this thesis has successfully demonstrated that temporal patterns can be successfully learned by machine learning programs and used to decrease execution speed, help make decisions and aid in context discovery.

Acknowledgements

I gratefully acknowledge my thesis advisor, Dr. Susan Epstein, whose guidance, inspiration, and knowledge have been invaluable these past years. She provided me with a stimulating work environment, ideas and direction for my research, and outstanding academic knowledge. Besides teaching me so much, she encouraged a professional and perfectionist attitude in my research and writing, from which I gained a lot. I also thank her for her understanding and support during the circumstances that life brought about in the years that spanned my PhD research.

I express my appreciation to the three other members of my thesis advisory committee, Dr. Jack Gelfand, Dr. Keith Harrow and Dr. Virginia Teller, for joining my committee and being supportive of my work. Special thanks to Dr. Jack Gelfand for the enlightening discussions I had with him in the early part of my PhD research, from which many ideas were born. A special mention also to Dr. Keith Harrow for his time and input during my final draft writing. I thank Dr. Stanley Habib for facilitating the fellowship and tuition support that I received in my years at CUNY, and Dr. Ted Brown for his understanding during my leave of absence and interest in my work upon my return. I express my appreciation to Mr. Joe Driscoll for being so helpful to me.

Above all, I thank my husband and my parents whose constant support and encouragement in every aspect of my life enabled me to reach this milestone.

Table of Contents

List of Tables

List of Figures

Chapter 1

1.1: Introduction

In the transition from novice to expert in a skill domain, practice plays a major role (Ericsson and Smith 1991). Each episode one experiences holds a wealth of information that, if extracted and utilized, may help one achieve a higher level of skill. One important kind of knowledge lies in the temporal patterns that occur in a domain. The word *pattern* here includes any perceived regularity in a set of data. A *temporal pattern* is a sequence of actions across time that creates the final pattern.

The thesis of this work is that the way patterns are formed over time can be learned during experience in a domain, and subsequently be used to improve decision making in that domain. Learning from temporal patterns can aid in the transition towards expertise, because it can both increase the speed of decision making and improve the quality of the decisions made. While many researchers have used patterns to aid decision making, and others have explored ways to learn the significant patterns in a domain, the idea of focusing on the order with which these patterns are formed is novel. Learning temporal patterns applies to any domain where important patterns exist, and where one can learn from experience.

This thesis investigates three approaches to learning temporal patterns in the domain of game playing. Here the definition of a pattern becomes more restricted: a *pattern* refers to a geometric group of markers or blanks on a game board. Chapter 1 of this thesis includes some background knowledge, related research, and an overview of the three learning methods developed in the course of this work. Each of the three subsequent chapters describes a learning method, its implementation, and experimental results with it. This thesis concludes with a discussion in Chapter 5.

Some general terminology is defined here. A *game* is an activity one can play, with a board, pieces and rules of play. A *contest* is an experience playing a game, from the first move until the rules specify a win, loss or draw. A *ply* is a move made by one player.

This chapter introduces this thesis' topic and its significance in Section 1.1. Section 1.2 discusses general artificial intelligence and game-playing background, followed by an in-depth summary of related game-playing research in Section 1.3. Section 1.4 introduces a computer game-playing program, Hoyle, the context for this work, and then provides the cognitive motivation for this thesis, and a short synopsis of the three learning methods researched. The chapter concludes with the games and terminology used in this thesis.

1.2: Background

1.2.1: Artificial Intelligence

Artificial intelligence *(AI)* attempts to program computers to perform various activities that normally require human intelligence. Four fundamental areas of AI research are knowledge representation, search, planning and learning. Before solving complex AI problems, one must represent knowledge about the relevant world in a way that can be manipulated by a program. Research in knowledge representation attempts to find representations that are representationally adequate, yet at the same time support both efficient inference and knowledge acquisition. Many AI methods use a state space representation. The *state space* for a task consists of the set of all states, representing all possible situations in the domain, and the set of operators that transform one state into another. Amarel's work on the missionaries and cannibals problem, and subsequently Anzai and Simon's experiments with the Tower of Hanoi problem, demonstrated how the

choice of an appropriate state representation can have a critical effect on problem solving (Amarel 1968; Anzai and Simon 1979).

Much of the work in AI views problem solving as search through a problem's state space for a goal state (Russell and Norvig 1995). Search begins at an initial state, and operators are applied until a goal state is reached. Because search with a state-space representation can degrade when there are many states, research eventually shifted from blind, uninformed search methods to heuristic methods that incorporate knowledge to reduce search.

A plan is a sequence of actions that leads from an initial state to a goal state. Planning differs from search-based problem solving in three ways. First, planners differ in their representation of states and actions. Planners usually represent states as sets of sentences in first order logic and actions as logical descriptions with their preconditions and effects. This enables a planner to choose relevant and worthwhile actions, that is, those actions that achieve a desired precondition. Second, planners can add actions wherever they are needed, rather than work incrementally from the initial state to the goal state. Third, planners often use the divide and conquer strategy, solving several subproblems and combining them to form one plan.

Machine learning constructs computer programs that automatically improve with experience (Mitchell 1997). Learning spans a broad range of processes, *from rote learning* (the simple storage of computed information) to *discovery learning*, where an entity must acquire knowledge without the help of a teacher. Machine learning research has concentrated both on methods that acquire new knowledge, and on methods that simply refine the knowledge a system possesses for it to be used more effectively.

1.2.2: Computer Game Playing

Game playing has long been a domain of interest to the machine intelligence community, because it addresses the four fundamental areas of AI mentioned above: knowledge representation, search, planning and learning. Furthermore, games provide a noise-free, structured framework where research can be done and measured for success before being adapted in other types of problems. Games have clearly defined rules and goals, and often possess large bodies of background knowledge that can be used to evaluate the level of expertise achieved by a game-playing program. Many important ideas have emerged from game-playing research, including certain search heuristics, reinforcement algorithms, evaluation function techniques, temporal difference algorithms, and pattern learning (Fuernkranz 2001).

Games also serve well as an experimental testbed for work in AI, because they are similar to real world problems (Michie 1974). Whereas in a puzzle, change effected by a given move is fully determined, in a game, a player has to take into account a range of possible consequences. A human expert player forms contingency plans against his opponent, much as a person does for obstacles in the environment. In the 45 years since Samuel's pioneering work on checkers, the game-playing domain has served as a rich domain for solving serious and difficult AI problems (Schaeffer 2000).

1.3: Related Work

1.3.1: Game Playing, Search and Knowledge

A game can be represented with a *game tree*, a search tree whose root is the initial game state. Each node in the game tree points to all its possible successor game states (one from each legal move). The *leaves* of the tree are the end nodes that have no successor

states. A *game state* includes a configuration of playing pieces on a game board and the identity of the mover.

The traditional AI approach to game playing is a "brute-force" approach, which performs a fast, deep search on the game tree. Most game-playing programs use the minimax algorithm to perform search and choose a move. The *minimax algorithm*, in theory, generates the whole game tree and applies an evaluation function *F* to the game states at the leaves of the tree. For any state S, *F(S)* is set to 1 or more if S is a winning state, -1 or less if S is a losing state, and 0 if S is a draw state. The minimax algorithm backs these values up the tree as follows. For each ply where the program is the mover, the value backed up is the one that maximizes *F*. For each ply where the opponent is the mover, the value backed up is the one that minimizes *F*. In principle the algorithm can produce perfect play because it can determine the *game theoretic value* of any game state (the true value of a state backed up from the leaves of the complete game tree). Exhaustive minimax analysis, however, is intractable in any large game tree. Most programs, therefore, search to some depth and then approximate the values of the *tip nodes* (the game states at the fringe of the search tree) with a *heuristic evaluation function* (a function that approximates F) (Shannon 1950). The values backed up are approximate because the algorithm no longer always searches to the leaves of the game tree. To reduce search even further, alpha-beta pruning eliminates from consideration branches that cannot possibly influence the final decision (Samuel 1959).

Minimax has several drawbacks. In a large space the evaluation function can often offer only a heuristic estimate of a game state. The procedure also chooses each move independently from the others. If the program is repeatedly presented with states where

there is more than one reasonable move, it might wander from one area of the game tree to another, rather than follow one line of play as human players do.

The role of knowledge in computer game playing has become more prominent over the past few decades. Early research in game playing restricted knowledge primarily to the evaluation function for board positions. Many games can be divided into three stages: the *opening* (the first portion of moves made), the *middlegame* (the moves between the opening and the endgame), and the *endgame* (the last stage of a game). When there are different objectives in each stage, different principles of play are used (Berliner 1979). Different evaluation functions for the various stages in a game permit programs to incorporate more knowledge into search (Samuel 1967; Berliner 1980; Lee and Mahajan 1990). The explicit representation of more knowledge reduces required search, and improves the quality of play. Large *opening books* (thousands of opening move sequences from contests played by humans) support game-playing programs. Whether they are provided as input or learned, it is easier to memorize these sequences than to evaluate nodes so high up in the game tree, i.e., at the beginning of a contest. In addition, *endgame databases* (positions that occur toward the end of the game, each associated with its proven game theoretic value) can provide perfect information. Because endgame databases may consist of billions of positions (e.g., checkers), much work is often done to compact the representation for real-time accessibility (Schaeffer 1997).

A prime example of a program which benefits from the above methods is Chinook, a world championship checkers program (Schaeffer 1997; Schaeffer 2000). Chinook, using alpha-beta search with numerous extensions, is able to search on average 19-ply into the tree. Its evaluation function is handcrafted and manually tuned. In addition, Chinook has

an extensive opening book and endgame database. Retrograde analysis was used to create the full eight-piece or less checker endgame database of about 444 billion positions. Besides improving endgame play, the endgame database also improves middlegame performance: whenever middlegame search reaches the endgame database, perfectly correct values from it are backed up the tree

The most successful chess program, Deep Blue, which defeated the human world champion, used alpha-beta minimax with various extensions and an expert-crafted chess evaluation function with more than 8,000 parameters (Campbell, Hoane et al. 2002). Other knowledge sources Deep Blue relied on were an opening book, an endgame database with five or fewer pieces on the board, and a large database of grandmaster moves used to encourage or discourage a line of play (Campbell 1999). Deep Blue's success can also be attributed to the hardware advances and special-purpose processing units that increased the search by orders of magnitude (Campbell, Hoane et al. 2002).

Shogi, or Japanese chess, is similar to chess, and computer shogi has benefited from adopting many of Deep Blue's ideas. The main reason shogi is more difficult than chess to program is because it has a significantly larger search space. Chess has an average branching factor of 35 and an average game length of 80, whereas shogi has a branching factor of 80 and an average game length of 115 (Matsubara, Iiada et al. 1996). Furthermore, shogi differs from chess in that it allows its pieces to be reused after they are captured. Because of this reuse rule, the number of possible moves (and states) increases from the middlegame to the endgame, rather than decreasing as it does in chess. Thus shogi cannot benefit from optimal play based on endgame databases (Iiada, Sakuta et al. 2002). Opening procedures are also difficult to build, because books written on

shogi opening theory discuss it in terms of patterns rather than move sequences. Although the differences between the two games force shogi programs to incorporate some novel ideas, their similarities lead shogi computer experts to predict the existence of a world champion shogi program based on game-tree search by 2010 (Iiada, Sakuta et al. 2002). However, as seen with chess, predictions in the past have been overly optimistic.

Despite AI's success with chess (and perhaps with shogi too), computer Go remains a challenge for AI. The traditional minimax approach to Go has produced only mediocre computer play, and is not enough to conquer this complex game (Muller 2002). Go's search space is significantly larger than shogi's search space, with an average branching factor of 250 and an average game length of 150. Unlike chess, where pieces are removed as the game progresses, in Go the pieces remain on the board. Thus Go is less amenable to endgame databases, because the greater the number of pieces on the board, the greater the number of possible states. Most important, there is no simple evaluation function for Go. Stones in Go do not have a mathematical value as in chess; rather their worth depends on the nature of the stones surrounding them (Muller 2000). Hence, the static evaluation function for Go is orders of magnitude slower than the one for chess. For these reasons, Go remains an open research domain and new approaches are being explored (Muller 2002).

1.3.2: Game Playing and Planning

Although long-range plans are typical of human game-players, most game-playing programs use the search-based paradigm and do no planning. After a person devises a plan to achieve a specific goal, s/he may search only the lines of play relevant to the plan. Without planning, game-playing programs examine search trees many orders of

magnitude larger than the trees searched by human experts (Russell and Norvig 1995). Planning research in game playing has been limited, however, because it is different and more difficult than planning in other AI domains. Planning in other domains usually relies on abstracted spaces, where unimportant detail is omitted to simplify the planning process. It is not clear how to define abstracted spaces for a game. Even a seemingly minor detail, such as a pawn in a certain location on a chessboard, may be important, and should not be abstracted. In addition, in other domains such as robot problem solving, the preconditions and effects of actions are well-defined, as is the test for having achieved a goal. In a game, preconditions for a move that should be taken are not clear-cut; very similar positions may require different moves. Furthermore, the effects of an action may be subtle, far-reaching and hard to determine.

One of the most successful attempts in game-playing planning research was PARADISE, a chess program that dealt with tactically sharp positions in the middlegame (Wilkins 1980). A *tactically sharp position* is one where success can be judged by *material* (the total value of a player's pieces on the board), rather than by a *positional advantage* (better piece structure). PARADISE possessed a database of chess knowledge in the form of production rules. Each production had a *pattern* (a complex interrelated set of features) as its condition, and an *action*, which consisted of (one or more high-level chess goals, e.g., CAPTURE and THREAT). The program performed a *static analysis* of a board position, matching patterns on the board to productions in the knowledge base to determine which concepts should be used in the reasoning process. PARADISE then linked these concepts to formulate plans that guided a small search tree for the next several ply to select the best plan. This expensive static analysis was executed

infrequently, when the program finished executing a plan. PARADISE produced trees of the same order of magnitude as those produced by human masters– hundreds of nodes, instead of the hundreds of millions of nodes considered by current search based programs. Because its search trees were so small, PARADISE was able to find *combinations* (important move sequences) as deep as 19 ply. PARADISE's main limitation was the amount of CPU time it required, due its inefficient production matcher. Furthermore, it never played a full game of chess; it was limited only to solving tactically sharp middlegame chess problems.

1.3.3: Game Playing and Learning

There are several reasons to use machine learning in computer game playing. First, while most game-playing programs rely on a real-valued evaluation function to estimate how close a state is to a goal, it is difficult for humans to hand tune the weights for the components in the evaluation function. Machine learning methods that automatically tune the evaluation function might replace humans in this task. A second reason for learning in game playing is to automate the selection of features for the evaluation function. Game-playing programs should also learn to accelerate and improve their behavior based on their experience. It seems wasteful for a program to handle a recurring situation in the same way, using the same amount of resources every time; previous experience should affect future problem-solving capabilities. A *transposition table* (an associative memory that stores recently evaluated positions) is an example of a learning method that accomplishes this (Samuel 1959; Greenblatt R. D., Eastlake III D. E. et al. 1967; Slate 1987).

A game player can learn to improve its play using a variety of methods. In 1961, Donald Michie built a simple learning machine, MENACE, out of matchboxes (Michie 1974). It learned to play tic-tac-toe using a primitive reinforcement method. The machine, which had prior knowledge only of the rules for legal moves, began by playing randomly and learned to play via practice. At the end of a game, if the machine won or drew, Michie rewarded all the moves the machine made during the game, to increase the probability that they would be chosen again. The terminal move received the greatest reward, while the moves taken closer to the beginning of the game received gradually discounted rewards. When the machine lost, negative reinforcement was applied in a like manner. Using this method, the machine learned to draw after twenty contests when Michie played a perfect strategy. When Michie resorted to unsound variations in an attempt to trap it, MENACE seemed capable of handling any situation after 150 contests. One limitation of Michie's method is that it stored every possible game state with its corresponding move and hence, would not work in much larger spaces. Also, it is not necessarily the case that the best or worst moves are found at the end of game. This *credit assignment problem* (how to determine which actions should get credit for a win or blame for a loss), has no simple solution (Minsky 1963).

Samuel wrote a checkers playing program that did both rote learning and polynomial evaluation learning (Samuel 1959). For rote learning, the computer stored in memory many game states and their corresponding computed evaluation scores, to reuse moves on previously seen states without reevaluating them. The saved time was used for deeper search, and enabled the program to improve over time. The number of game states saved was limited by deleting *redundancies* (repeated states), and reporting all states as if it

were black's turn to move. Samuel also aged each state, *refreshing* it (dividing the age by 2) when the state was used, and *forgetting* it (dropping it from memory) if the age went above a certain value.

For polynomial evaluation learning, the program learned which 16 features of the checkers game state (out of 38 preselected by Samuel), should be used in the evaluation function, and what their coefficients should be. The program learned through *self-play*, where two versions of the program, Alpha and Beta, played against each other. Whereas Alpha generalized on its experience after each move by adjusting its coefficients and replacing terms, Beta used a single evaluation function for the entire contest. When Alpha won a contest, its evaluation function was assumed to be better than Beta's, and its current evaluation function was given to Beta. If Alpha lost three times to Beta, this indicated that Alpha was going in the wrong direction, and the coefficient of the term with the largest coefficient was set to zero in an attempt to set it back on track. Alpha updated the weights of its parameters by looking at the difference between the static evaluation score of its current position and the backed up score obtained by minimax search. If the search returned a significantly higher value than the static evaluation function, then the weight of each positive feature in the evaluation function was increased, and the weight of each negative feature in the evaluation function was decreased. The opposite was done when the search returned a lower value than the static evaluation function. After playing against itself thousands of times, Samuel's program learned to play a "better-than-average" game of checkers, better than Samuel himself.

In subsequent research, Samuel improved the expertise of his checkers player with two major changes (Samuel 1967). Previously the program learned a linear evaluation

function that failed to account for interactions between features. Samuel added *signature tables* (multi-dimensional tables where each dimension represented a feature) to handle nonlinearity. The values of the interacting features were used to index into the table to extract the combined value of these features. To reduce the size of the tables, Samuel organized them hierarchically. In addition, instead of learning via self-play, the new version of the program learned to emulate master play from contests played by expert checker players. While the second version still did not reach master-level play, it greatly improved the program's performance.

TD-Gammon used a temporal difference method with network learning to tune the weights for a backgammon evaluation function (Tesauro 1995). Early versions of the program used a database of backgammon states as a training set; later versions learned from self-play. After playing about 1,500,000 self-play contests, TD-Gammon 3.0 was able to learn weights for a three-layered, fully connected feed-forward network that played world-championship-level backgammon.

Lee and Mahajan used Bayesian learning techniques to tune an Othello evaluation function from a training set made up of game states found in contests played by two expert players (Lee and Mahajan 1990). The advantage of their approach was that it automatically computed the variance and covariance of features, allowing it to learn a non-linear, rather than linear, evaluation function. Their program, BILL, played Othello on the world championship level. Unlike Samuel's program, which used sixteen features in its evaluation function, Lee and Mahajan's program used only four. They chose to use a small number of computationally expensive features rather than a large number of features that were cheaper to compute.

All the programs discussed above are initially provided with a set of features for which they must learn weights. Currently, an important area of research is constructive feature generation: how to learn the feature set that should be used for the evaluation function. Much of the work in this area has been done with artificial neural networks, where various algorithms are used to add new units, or to split a unit in the network under certain conditions. For example, *Dynamic Node Creation* checks the size of a network's error whenever the error asymptotes (Ash 1989). If the error is very large, the program adds a new hidden unit to the network. Another network example, *node-splitting*, takes a hidden unit and splits it into two, if the unit's hyperplane is oscillating (Wynne-Jones 1992). A different approach is taken by ELF, a non-linear function approximator for real-valued functions over a space of boolean-valued variables (Utgoff and Precup 1997). ELF uses a complete pattern representation that can represent any possible feature. A new feature is constructed by specializing an old feature that enables the weight adjustments to reduce error. The disadvantage of this approach is the vastness of the feature space, which must be searched before an appropriate feature set is found for a particular problem.

1.3.4: Game Playing and Patterns

De Groot was one of the earliest researchers to recognize the importance of familiar patterns, by which he meant groupings of pieces meaningful to chess masters. In a comparison study between masters and weaker players, de Groot could not find any quantitative differences that seemed to underlie chess skill (De Groot 1966). The gross thought processes and the search depth of both master and weaker players seemed similar, yet masters were able to hone in on stronger moves, while the weaker players

invariably wasted time exploring weaker moves. De Groot did find, however, that masters were much better at reconstructing chess positions to which they were briefly exposed, a skill that he attributed to their ability to perceive familiar patterns of pieces on the chessboard.

Chase and Simon conducted further chess experiments in the early 1970's and, in conjunction with de Groot's experiments, suggested that the ability to perceive patterns of pieces is not a side effect of chess skill, but is in itself one of the important processes underlying chess skill (Chase and Simon 1973). Their chunking research seemed to indicate that chess masters had a large pattern repertoire available to them that was built up through practice. According to Chase and Simon's theory, when playing a game, chess masters scan the board to find familiar patterns, which are then used to suggest the plausible moves. This theory became widely accepted and influenced much of the subsequent research on chess and game playing in general. Many programs and systems were written in an attempt to simulate this *recognition association* theory, or at least to make use of patterns as a knowledge resource while playing a game. Although subsequent research argued that this theory is incorrect, pattern recognition remains one of the tools used by grandmasters to prune search and evaluate positions (Holding 1985).

One successful system that incorporated pattern recognition ideas was HiTech, a chess program built upon the SUPREM architecture (Berliner and Ebeling 1989). Although, like most chess programs, it searched deep into the tree, SUPREM differed in its use of pattern recognition as its evaluation mechanism. The architecture contained a chess oracle with production-like rules that defined goals and the patterns needed to achieve them. The oracle, coded by chess experts, was intended to be as perfect as the

experts could make it, and contained 20,000 lines of code with 40 downloadable patterns. The oracle performed a detailed analysis of the root position, to determine which of the available patterns in memory should be downloaded into recognizers for the current search. At the end of the search, the recognizers found the patterns that were present on the boards at the tip nodes, and used them to aid in the evaluation of these boards. The main limitation of this approach was that it did not learn; its success was dependent upon its built-in pre-specified patterns, and upon the ability of the oracle to choose the correct patterns to download at each point.

One of the strongest Othello playing programs, Logistello, also relies upon important patterns (Buro 1997). In its original version, the system designer provided the important patterns and the program learned an evaluation function that used these patterns as features, along with some simple parity features. It trained on a set of example game states from well-played contests. Later, Buro developed GLEM, which created a much more sophisticated evaluation function by linearly combining simple binary features. Although GLEM approximated highly complex features, ultimately, even GLEM requires predefinition of the atomic features (Buro 2002). Buro also developed Probcut, a selective search algorithm which prunes the alpha-beta search, and an opening book learning algorithm which enabled Logistello to avoid losing in the same way twice and explore new opening variations (Buro 1995; Buro 1999).

In many domains, however, the important patterns are not known in advance. Ideally, a program should be able to learn important patterns from experience, just as a person does. An early attempt to learn about patterns and plans was made by Minton in 1984 using a constraint-based generalization method (Minton 1984). Minton developed a

program that learned game playing plans for tic-tac-toe, chess, and go-moku, (not to be confused with Go), from single examples where the opponent achieved a specific goal state. The program continuously regressed from the goal pattern as long as its move was forced, to find the original pattern that led to the winning sequence. This method was limited to learning plans only for forcing states that led to a win, and could not discover other interesting patterns.

An example of a program that made such an attempt is Morph, a chess program that learned threat and defense graph patterns to use as features in its evaluation function (Levinson and Snyder 1991). After each contest that Morph played, patterns were created, deleted, and generalized, and pattern weights were updated to make Morph's evaluation more accurate. Even though Morph learned many correct pattern-weight associations, it had limited success. Morph learned to defeat human chess novices, and to beat its low-level Gnuchess trainer, but did not improve much after that. Morph's authors believed that it lacked appropriate mechanisms for combining the weights recommended by individual graph patterns, thereby failing to consider the relationships among patterns. Another weakness was Morph's inability to have information learned about one pattern influence the value of similar patterns. Morph II was a domain-independent generalization of Morph that used a graph representation for both game rules and patterns (Levinson 1996). It combined pattern values hierarchically to overcome Morph's greatest weaknesses. Morph II, however, was used only to implement simple games like tic-tac-toe and nim, and was never extended to chess.

Morph III and Morph IV, extensions of the original Morph chess program, used four knowledge sources in their evaluation function: two conventional ones, material and

mobility, and two new ones, distance (based on shortest safe paths between two pieces) and nearest neighbor classification (Allen, Hamilton et al. 1997; Levinson and Weber 2000). Every state encountered was abstracted to form a pattern vector and stored in the nearest neighbor classifier (NNC) table. A pattern vector described the state in terms of the attack and defense edges between pieces on the board. During play, the NNC looked for the pattern in the table that was most similar to the current state's pattern vector and assigned the evaluation value of the matched pattern to the current state. This value was then combined with the other three knowledge sources to form the final evaluation for the state. Although Morph III and Morph IV significantly outperformed the original Morph system, they required large amounts of training data to achieve a fair level of play. In subsequent research, a new graph representation called chess neighborhoods produced a program that, after a few days of training on the Internet Chess Club, achieved the same ranking (1024) as Morph IV did after a few months (Levinson and Weber 2000).

Ginsberg, in his overview of computer game-playing programs, describes how Go is a fundamentally parallel problem that involves a lot of intuitive, human-like pattern matching abilities (Ginsberg 1998). In Go, human players recognize not just patterns of stones and empty spaces, but much larger perceptual and relational concepts between groups of stones. Almost all Go programs use a database of patterns to encode Go knowledge (Muller 2002). Early on, Reitman and Wilcox tried to imitate human perceptual and cognitive abilities with a cognitive computer Go model (Reitman and Wilcox 1979). The current best programs, however, such as David Fotland's Many Faces of Go, use a much simpler pattern matching approach. Many Faces of Go has a hand-generated database of approximately 1,000 8x8 patterns used to aid in move selection

(Kojima and Yoshikawa 1999; Burmeister 2000). Patterns are necessary to select the best possible moves since it is computationally impossible to thoroughly consider all possible legal moves as programs for other games do.

While most current programs use hand-generated pattern databases, machine learning techniques are being researched to extract high-quality patterns from professional Go games (Muller 2002). Recently Kojima and Yoshikawa have researched two approaches to automatically acquire pattern knowledge from professional game records, for the Game of Go (Kojima 1998; Kojima and Yoshikawa 1999). The first was a deductive approach used to acquire strict knowledge whose validity is easily provable. The system began with one forcing rule that dealt with capturing stones, and learned new rules by backtracking and using Explanation Based Generalization whenever a known forcing rule was applied (Mitchell, Keller et al. 1986). Although the system acquires proper rules, the rules deal only with the goal of capturing stones, and cannot acquire other interesting rules where the goal is vague and not easily explained.

The second approach was an evolutionary approach used to acquire large amounts of heuristic knowledge from a large number of training examples, whose effectiveness is hard to show, but whose knowledge is much more flexible (Kojima 1998; Kojima and Yoshikawa 1999). Because the pattern search space is intractable, the authors felt that an evolutionary approach would approximate solutions without exhaustive search. The system acquired useful rules of a variety of sizes and shapes, unlike former algorithms that acquired only fixed-shaped patterns. However, the conditions of the patterns included only stones, edges, and previous moves, without considering more human-like conceptual knowledge such the number of liberties (i.e., vacant intersections adjacent to

stones on the board). The evolutionary algorithm was applied to Tsume-Go (a subset of Go problems), and comparison with cognitive experiments that tested humans on Tsume-Go showed the system to be as good as a 1 dan amateur human player. Interestingly, results of the cognitive experiments that tracked human eye-movements confirmed that humans, like the evolutionary system, used knowledge and patterns rather than search when solving Tsume-Go problems (Kojima 1998).

1.4: The Context: Hoyle

FORR (FOr the Right Reasons) is a cognitive architecture for learning and problem solving that models the transition from general to specific expertise (Epstein 1994a). A FORR-based program is intended to learn in a broad domain of related problem classes. *Hoyle*, a FORR-based program, is an example of a game-learning program that learns to play through experience. Hoyle's domain is two-person, perfect information, finite-board games. A problem class for Hoyle is any specific game, such as tic-tac-toe. A FORR-based program uses a set of domain-dependent, hierarchically-organized Advisors to make its decisions. Each Advisor has a narrow viewpoint of the world, and gives advice based on this viewpoint. For example, in Hoyle, Advisors represent general game playing principles. A full list of Hoyle's Advisors can be found in Appendix B. When it is FORR's turn to make a decision, the Advisors have access to the current state of the world, the current useful knowledge collected, and the set of legal actions. Because some domains have states that are equivalent for decision-making purposes, FORR permits each program to normalize such states. Hoyle normalizes with the eight-element group of symmetric transformations on the two-dimensional plane.

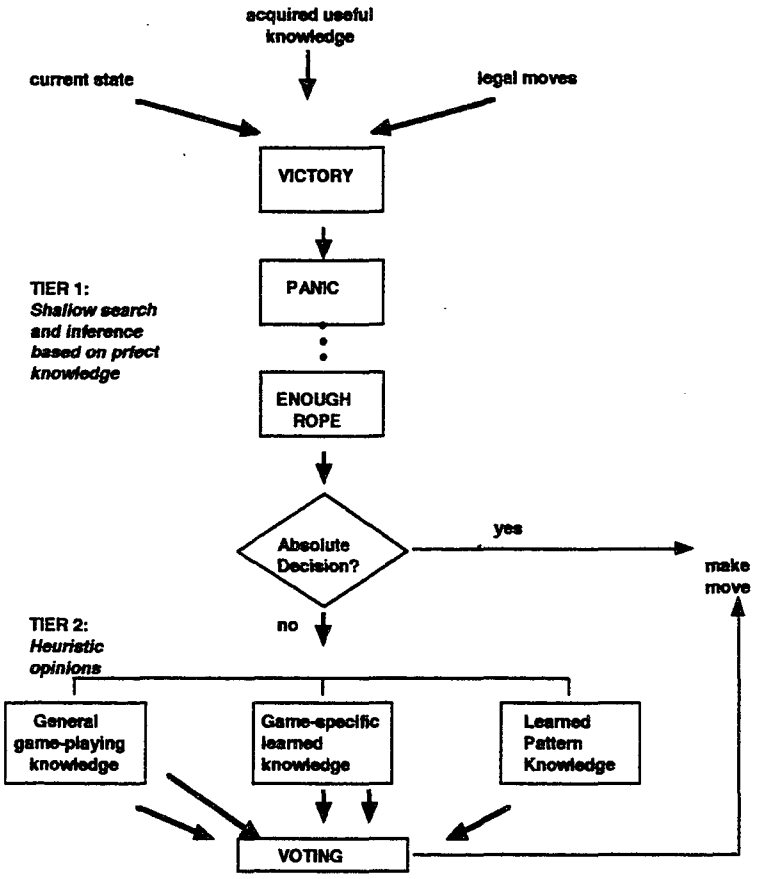


Figure 1: How Hoyle makes decisions

Decision-making control in Hoyle flows through the two-tiered hierarchy of Advisors in Figure 1. The first tier of Advisors is consulted sequentially in a pre-specified order. These Advisors are perfectly correct and attempt to reach a decision using shallow search and inference based on perfect knowledge. Some tier-1 advisors have *absolute authority* (a decision reached is executed without further deliberation). Other tier-1 Advisors eliminate certain moves from consideration and pass the remaining moves to the next Advisor in line. If a decision is not made in tier 1, control flows to the second tier of Advisors. These Advisors offer heuristic, possibly conflicting advice as comments, in the form <Advisor-name, action, strength>. Strength is an integer between 0 and 10,

denoting a range from strong opposition to heavy support for the indicated action. The action with the most support is selected as the decision.

A FORR-based program begins with a set of problem-class-independent, *useful-knowledge items*, each of which specifies something important to learn in this domain. Each item has a learning algorithm to acquire its knowledge, and a trigger that tells when to execute the algorithm. As the program gains experience executing tasks within a particular problem class, it acquires problem-class specific knowledge for the useful knowledge item. Although the Advisors themselves do not learn, many of them reference the acquired useful knowledge. Hence, over time, as the program gathers more knowledge, the Advisors' comments improve. For example, when learning to play a new game, such as tic-tac-toe, Hoyle begins with game-independent, general game-playing Advisors, a set of game-independent, useful-knowledge items without values, and the rules of the game. It begins by playing the game correctly, and plays more expertly as it acquires and exploits the game-specific knowledge that instantiates the useful-knowledge frame.

There are three major differences between Hoyle and most other game-playing programs. The first is that Hoyle learns to play many games expertly; it is not restricted to playing only one game. The second difference is that Hoyle not only learns, but it also successfully integrates multiple learning methods in one system. The third, and most important, way Hoyle differs is that it never searches more than two-ply deep during competition, unlike the extensive search that categorizes other game-playing programs. Hoyle does not possess an evaluation function, per se, although the collective advice of the second tier can be seen as one. Hoyle is an attempt to model the way humans play,

and learn to play, not an exercise in the construction of a champion at a specific game with game-specific knowledge.

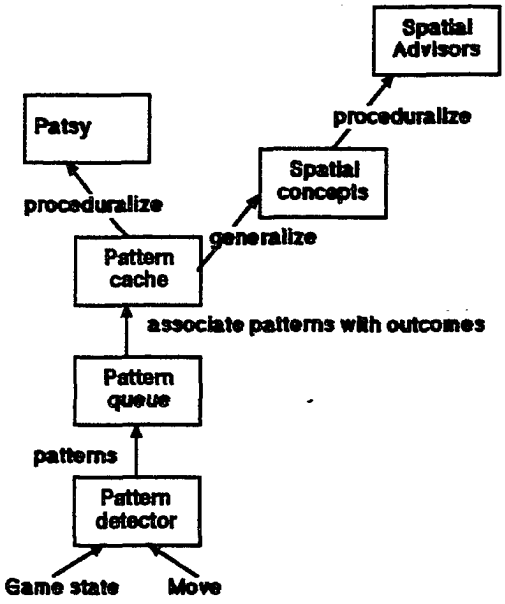


Figure 2: Hoyle's model for pattern learning

A recent addition to Hoyle is that of pattern learning capability and spatial-concept formation from the patterns learned (Epstein, Gelfand et al. 1996). An overview of Hoyle's pattern-learning model (bottom to top) is shown in Figure 2. Not all patterns are perceived and learned by Hoyle. Hoyle uses *BPL* (Bounded Pattern Language), a set of pre-specified, game-independent shapes: straight lines, squares, diagonals, L's, and triangles without right angles. The first time Hoyle encounters a game, it calculates the game's *board-specific templates*, which specify the location of the BPL shapes on the game board and their size in *metric units* (the smallest Euclidean distance between any two positions on a game board). Figures 3(a) and 3(b) display the templates and metric-units for two games. A pre-specified *field of view parameter* indicates the maximum permissible breadth in metric units for any board specific template. A pattern for Hoyle,

as shown in Figure 3(c), is an instantiation of a board-specific template with markers, blanks and don't cares (#'s). Figure 3(c) represents a pattern where White controls the corners of the outside square. BPL, the field of view parameter, and normalization for symmetry all restrict the number of patterns perceived by Hoyle.

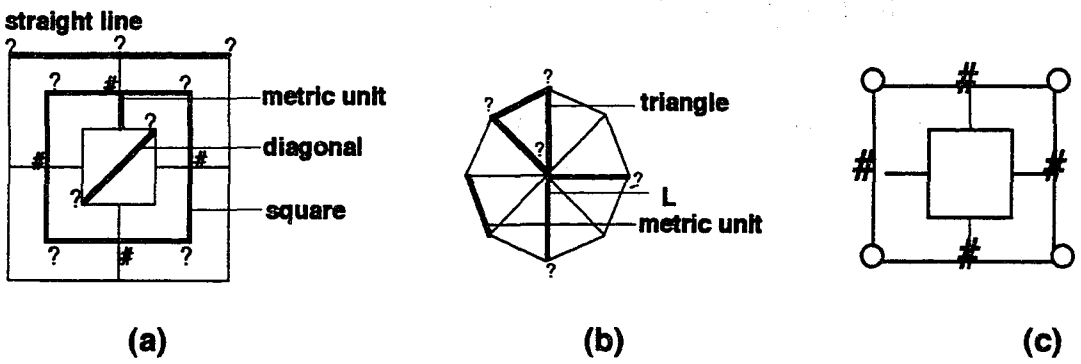


Figure 3: Templates for (a) nine men's morris and (b) shisima. (c) A pattern in five men's morris. # denotes "don't care."

After each contest played by Hoyle, the patterns created by each move are extracted. These patterns are stored in the *pattern queue* which contains three counters for each pattern: W, L and D, the number of times the pattern was seen in a game where the first player won, lost or drew, respectively. Patterns associated with a single *consistent* (only one of the counters is non-zero) outcome migrate to the *pattern cache*. Pattern knowledge is proceduralized in two ways. First, the patterns in the cache serve as input to a game-independent, tier-2 Advisor, Patsy. *Patsy* computes a value for each pattern in the cache based on its W, L and D values, and the total number of contests won, lost or drawn since the pattern was seen. The strength of Patsy's comments for each next possible legal move is a function of the values of the patterns found in the state to which the move leads. The second way pattern knowledge is proceduralized is via the construction of new, game-

dependent spatial Advisors based upon concepts formed from patterns seen in the cache. Periodic sweeps of the cache are made, to group patterns that share similar properties. Groups of patterns are then generalized with built-in generalization rules to form spatial concepts. Each of these concepts, represented as patterns on boards, is integrated into Hoyle's second tier as an individual spatial Advisor. A new spatial Advisor advocates or discourages a move that creates or destroys its pattern, depending upon whether it is considered to be a good pattern or not. Once these new Advisors are created, *PWL* (Probabilistic algorithm for Weight Learning), determines how well the Advisors emulate expert behavior (Epstein 1994b). PWL validates the accuracy of individual Advisors and eliminates those that prove irrelevant, untrustworthy, or contradictory.

Experiments done with tic-tac-toe, lose tic-tac-toe, and five men's morris have shown that Hoyle learned the same pattern-based concepts on every run of the game, despite the non-determinism of the program and its opposition. Furthermore, the concepts learned are those considered important in the game, and some of them were previously inexpressible in Hoyle's representation. In the past, pure pattern learners have learned too slowly, have learned too many patterns, and have proved unreliable. The idea behind the development of Patsy and the new spatial Advisors, is that pattern learning is an integral part of high-level reasoning, but not the sole factor in the development of expertise.

Although Hoyle learns to play some games well, it has some important limitations that must be addressed. Recall that Hoyle uses a built-in pattern language, BPL, to generate game-specific templates for every game it plays. Its pattern-oriented Advisor Patsy only notices and learns about the patterns covered by these templates. The rationale and validation for BPL are cognitive; studies have shown that people tend to recognize

regularly shaped patterns more easily, perhaps because they have built-in visual recognizers for them, like the templates that BPL produces (Treisman 1986). As a person develops expertise in a domain, however, s/he begins noticing new, more irregularly shaped patterns that prove, inductively or deductively, to be important to this domain. New visual recognizers are built for these important shapes, and they enter the perceptual process. Currently Hoyle has no ability to extend BPL or to notice patterns not covered by BPL.

Another limitation of Hoyle's pattern-learning capability is that it acquires only static pattern knowledge. After each move, Patsy looks at all the patterns covered by the template set and formed by this move, and updates its knowledge about these patterns. A *static pattern* is such a completely formed pattern found on the board at any point in time. Sometimes, however, the order in which a static pattern was formed is crucial information. Hoyle has no method of acquiring this type of temporal pattern knowledge, i.e., knowledge about how a sequence of moves forms patterns. Because of this, Hoyle is limited in the ways it can use its pattern knowledge. When it is Hoyle's turn to play, Patsy looks at the patterns formed by each possible move, and evaluates each move based on the goodness of the patterns it forms. The process uses a lookahead of one move; Hoyle has no way to suggest a series of moves to achieve a pattern. The patterns learned by Hoyle are examined every ten contests for groups that might form more general spatial concepts. These concepts are integrated into Hoyle's second tier as spatial Advisors that support or oppose moves that create or destroy its pattern, depending upon whether or not it is considered to be a good pattern. Once again, this method considers only patterns that can be formed on the next move, without looking further ahead.

As humans develop expertise at a specific game, their play speeds up, so much so that it sometimes seems as if they "play without thought." Many have explained this phenomenon as a shift from searching to pattern-oriented play. Hoyle, on the other hand, sometimes slows down in its transition towards expertise, even as it gathers more pattern knowledge. This could be because Hoyle adds its pattern knowledge to the decision making process, instead of sometimes shifting to a completely pattern-oriented decision. Alternatively, it could be because each state is considered as a separate entity, both when learning pattern knowledge and when applying it. Hoyle does not learn, represent and use knowledge about sequences of moves that belong together. Each method in this thesis takes a different approach towards learning sequence knowledge and making decisions using temporal patterns. The patterns learned are not restricted to a predetermined language such as BPL.

1.5: The Approach: 3 Methods

The development of a method to learn temporal patterns was inspired by experiments done by Chase and Simon in the early 1970's (Chase and Simon 1973). One experiment compared the memory for contests of masters, Type A (mediocre) players, and novice chess players. Each subject was asked to memorize a contest and was then tested for his recall of the contest on five subsequent days. Because the better players both learned quicker and recalled better, the experiment clearly demonstrated that chess skill influences memory for contests. Of further interest was that the players retrieved the sequences of chess moves in bursts, separated by larger pauses. In another set of experiments, the players were asked for immediate recall of 10-move sequences, where each move consisted of a player's action and the opponent's response. Once again, the

higher the level of chess skill, the better the recall, and parts of the sequences were again recalled together rapidly. These experiments provide clinical evidence that better human chess players have some *long term memory* (information that is retained in the mind for long periods of time) structure of move-sequences which they use to aid in their choice of moves. In other words, players do not merely learn from practice which patterns are important to achieve; they also learn the correct sequence to use to create each pattern. The methods I developed simulate this idea, and build up such long term memory *(LTM)* structures that can be used later in play.

Human game players learn and retain many move sequences from playing. It is unclear, however, which sequences are learned and how they are represented. According to Chase and Simon, verbal protocols indicate that players often retrieve these move sequences based largely on spatial and perceptual chess relationships (Chase and Simon 1973). Accordingly, one can postulate that the memory for sequences stores boards or partial boards and the changes made to them. As for which sequences are learned, I believe that a player will store sequences that have commonly arisen in play, are likely to reoccur, and have proved "safe," that is, led only to draws or wins for their player. In a game, one cannot predict with certainty what one's opponent's response will be in a particular situation, so one should not simply string together one's own and one's opponent's moves as a plan. There are times, however, that the opponent is likely to be dependable, and will probably reply in a certain fashion. These are the sequences that deserve to be stored in long-term memory, and extracted for reuse, to avoid unnecessary, expensive recomputation.

The goal of this thesis is to learn about how patterns are formed across time, rather than focus only on static patterns. Temporal patterns are complex to learn. It would be prohibitive to look at all possible sequences and the patterns they form. In addition, it is difficult to determine which of the pieces already present on the board are relevant to a specific sequence of moves. Learning should not consider the pattern formed only by an isolated sequence; it should address the context where the sequence was executed. Three methods are examined here; all of them are "forward" methods in that they learn temporal pattern knowledge from sequences of moves that create various patterns. Specifically, they do not consider every kind of pattern, since the moves that form many patterns may be spread out over a game and cannot easily be connected. Rather, the three methods learn about those patterns formed by a sequence of consecutive moves made by two players, or a sequence of consecutive moves made by one player (ignoring the opponent's moves). Just as expert human players have many opening sequences that they play by rote, there are other moments in a game where occasions for stereotyped sequences of play arise and are used. The two players move, one and then the next, as if in a ritual dance, building up a pattern with their pieces. The methods studied here attempt, in a forward manner, to identify these sequences, learn when they are appropriate, and store them for later use.

TP-Rote (Temporal Patterns by Rote) is a rote-learning, caching scheme that essentially hones in on frequently-used segments of a search space and memorizes them to reuse them later. It accomplishes this in a two-step process: gathering and filtering. First, action sequences are gathered from successful experience (i.e., from contests in which a particular contestant won or drew) and stored together with the situation where

they arose. Since it would be prohibitive to look at all possible sequences and the patterns they form, the method collects only short sequences. In the second step, *filtering*, from short action sequences that have proved important, TP-Rote builds up a *sequence tree*, a compact representation for longer action sequences. Whenever applicable, Hoyle uses sequences from these trees in further play. Experiments with TP-Rote indicate a significant speedup in Hoyle's play, simulating the gradual shift to "play without thought" seen in human game players at various points in a game. TP-Rote's limitation is its inability to generalize beyond symmetry, which limits sequence usage to situations that have already been seen. Chapter 2 investigates TP-Rote.

The second method, *TP-Context* (Temporal Patterns through Context) generalizes states, retaining only the *context* (the motivating description) of a sequence. Such a generalization may cover many states and thereby expand the applicability of a temporal pattern. This is especially important in large state spaces, where identical states may not often recur. Context discovery is a difficult AI problem. How can one determine which part of the board caused an individual move or a sequence of moves to be executed? Rather than deduce all possible contexts from the rules of a game or inspection of the search space, TP-Context exploits the knowledge inherent in experienced sequences to aid context discovery. It organizes its domain experience by associating a sequence of actions with the set of states where the sequence's execution began. Breaking the problem down into small clusters narrows the perspective, so that a context can be generalized from the small group of states that led to the same pattern of activity. A learned context is paired with its sequence to form a <context, sequence> pair, where the satisfaction of the context's conditions indicates that the sequence is a plausible course of

action. Given their inductive origin, however, the context pairs should not be used immediately in problem solving; care is needed to filter out the correct pairs from the incorrect ones. Using a perceptron-like weighting algorithm, each context pair is weighted in further play according to how well it emulates expert behavior. Highly-weighted pairs are integrated into the decision-making process. Chapter 3 investigates TP-Context.

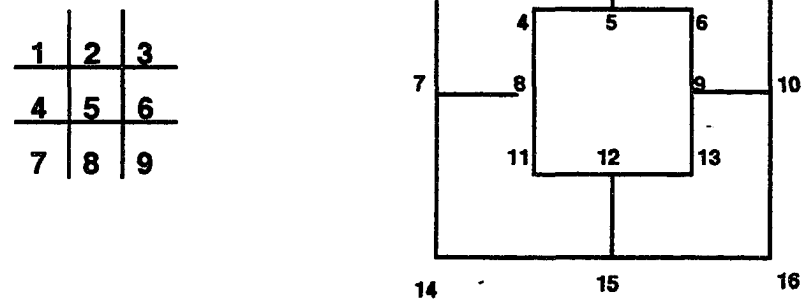
The third method, *TP-Sitact* (Temporal Patterns through Context using a situation-action representation) is actually an extension of TP-Context. It begins with the <context, sequence> pairs learned by TP-Context. Then TP-Sitact executes each sequence upon its context, to develop a specific context for each action in the sequence. Hence each <context, sequence> pair spawns a few corresponding <context, action> pairs, where the satisfaction of the context's conditions indicate the plausibility of an individual action rather than a sequence of actions. Like TP-Context, TP-Sitact weights the <context, pairs> in further play, and integrates the best ones into the decision-making routine. This representational shift was made to determine whether a sequence of actions or an individual action provides better advice. Results show that TP-Sitact leads to better performance than TP-Context, because the <context, action> pairs can be weighted more accurately than the <context, sequence> pairs. It is important to note, however, that the <context, sequence> pairs had to be created in order to acquire these <context, action> pairs. Furthermore, because TP-Sitact spawns many more context pairs, it requires significantly more execution time than TP-Context. Chapter 4 covers TP-Sitact.

1.6: The Games

Throughout this thesis, experiments were conducted on two games: *lose tic-tac-toe* (where the first to place three markers in a row on the board in Figure 4(a) loses), and *five men's morris.* Although lose tic-tac-toe is played on the same board as tic-tac-toe, lose tic-tac-toe is a much more difficult game to learn, both because the object is to avoid a certain pattern rather than to achieve it, and because a non-optimal move is often a fatal error (Cohen 1972). In five men's morris each contestant has five markers, which they take turns placing on the line intersections of the board shown in Figure 4(b). Once the markers are all placed, the contestants slide their pieces from one spot on the board to an adjacent spot. A contestant who achieves a *mill* (three owned pieces in a row) removes one of the opposition markers from the board. The first contestant reduced to two markers or unable to slide, loses. Five men's morris is significantly more challenging than lose tic-tac-toe; there are approximately seven million positions in its game graph. The full rules for both these games can be found in Appendix A. The hardware and software used for all experiments in this thesis can be found in Appendix A too.

In the following chapters, the symbol "#" is used to denote a "don't care." *Player-1* refers to the player that is first to move in a contest, such as X in lose tic-tac-toe, while *Player-2* refers to the player whose turn is second, such as O in lose tic-tac-toe. A place on a game board where a contestant's marker may be placed is called a *location.* All locations on a game board are numbered in row order, beginning with 1, as displayed in Figure 4. A *placing move* is one where a player takes one of his pieces not currently in play and chooses a location on the board to place it on. A *sliding move* is one where the player slides a piece already present on the board to another location on the board. A

draw game is one where the value of the root of the game tree is a draw under full minimax.



(a) (b)

Figure 4: The boards and their location numbers for (a) lose tic-tac-toe and (b) five men's morris.

Chapter 2: TP-Rote

This chapter examines TP-Rote, a learning method that learns temporal patterns by rote. These temporal patterns are added to Hoyle, enabling Hoyle's execution speed to decrease with expertise. Section 2.1 describes the TP-Rote learning algorithm, while Section 2.2 details the representation and implementation. Section 2.3 contains a description of the experimental design, followed by the experimental results in Section 2.4. TP-Rote is a learning component added to Hoyle, that learns to decrease Hoyle's execution speed as Hoyle gains expertise. The chapter concludes with a discussion in Section 2.5.

2.1:The Algorithm

TP-Rote (Temporal Patterns by Rote) memorizes commonly occurring sequences of actions engaged in by two agents. In game playing, the agents are the contestants, an action is a legal move, and a state is uniquely described by the game board and whose turn it is to move. While learning to play a game, TP-Rote associates a state with short paths in the search space that followed the state during play, and identifies those paths that were often used. From these short action sequences that have proved important, TP-Rote builds up a compact representation for longer action sequences, called a *sequence tree*. Sequence trees are stored in a hash table keyed by the state description.

Figure 5 illustrates a simple example of a tree learned for X in *lose tic-tac-toe*, (played like tic-tac-toe except three in a row loses). When the key is the board in Figure 5(a), the value returned is the tree in Figure 5(b), which represents two sets of moves that often arise in this game, ones that Player X should learn to play automatically. The tree indicates that X should play in 9, and if O responds in 6, X should reply in 4, and then if

O plays in 2, X should respond in 8. Alternatively, if in response to 4, O plays in 3, X should respond in 7. Essentially TP-Rote hones in on segments of the game tree that are often used. Instead of storing the full game tree, or storing all boards experienced with their corresponding moves, both of which require too much space, TP-Rote builds up a condensed hash table where each entry for a complete state description is a small tree of moves. This representation is referred to as *situation-sequence representation* (a state description is matched with a sequence of decisions), rather than the typical *situation-action representation* (a state description is paired with an individual action) used by many machine learning applications (Samuel 1959; Levinson and Snyder 1991; Finkelstein and Markovitch 1998).

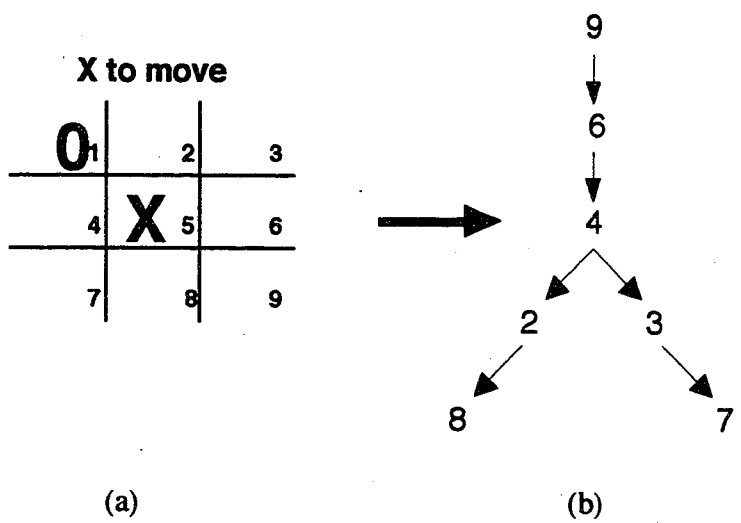


Figure 5: A lose tic-tac-toe sequence tree learned by TP-Rote (a) the state with location numbers (b) the sequence tree

2.2: Representation and Implementation

2.2.1: Building a Sequence Hash Table

After every contest, TP-Rote collects the sequences of moves that occurred in the contest and stores them in an *SHT* (Sequence Hash Table). The index into an SHT is a state; the value stored for the index is a list of sequences that were executed on that state in play. Sequences of all possible lengths would consume too much space, but it is difficult to determine *a priori* what length sequences to collect. In different games, and even within one game in different situations, there may be sequences of various lengths that are important. TP-Rote avoids this problem by collecting only small sequences, and building up larger ones from the smaller ones. This controls the total possible number of sequences collected, but still permits sequences of various lengths. Initially, TP-Rote collects sequences of length 2 and length 3. Using these as building blocks, any even length sequence can be formed by concatenating length 2 sequences, and any odd length sequence can be achieved by concatenating an even number of length 2 sequences with an odd number of length 3 sequences. TP-Rote builds up two SHT tables: one for states where the first agent, i.e., Player-1, is the actor, and one for states where the second agent, Player-2, is the actor. Sequences are gathered only from contests in which the featured player won or drew. For example, Player-1 sequences are collected only if Player-1 won or drew.

To focus on the responses of the agent whose sequences TP-Rote is learning, even-length sequences begin with the other's action and odd length sequences with one's own action. For example, when TP-Rote learns sequences for X in lose-tic-tac-toe, it collects all length-2 sequences of the form <O's action, X's action>, and all length 3 sequences of

the form <X's action, O's action, X's action>. Likewise, when it learns for O, it collects all length-2 sequences of the form <X's action, O's action> and all length-3 sequences of the form <O's action, X's action, O's action>. Each sequence is then recorded in the SHT on the value list of the state it was executed upon. For example, Figure 6(b) shows a length-3 lose-tic-tac-toe sequence recorded for X as a response to Figure 6(a). If, later in play, another sequence was executed upon Figure 6(a), the new sequence would be added to the state's value list of sequences. States in an SHT are normalized for the symmetries of the two-dimensional plane, and the recorded sequences correspond to the normalized states. This permits the application of a symmetrically equivalent version of the sequence (here < X to 4, O to 8, X to 2 >) to a symmetrically equivalent state such as Figure 6(c). TP-Rote also maintains counters for the number of times the state arose in play (*state-seen*), and the number of times a particular sequence was executed (*sequence-seen*) from that state.

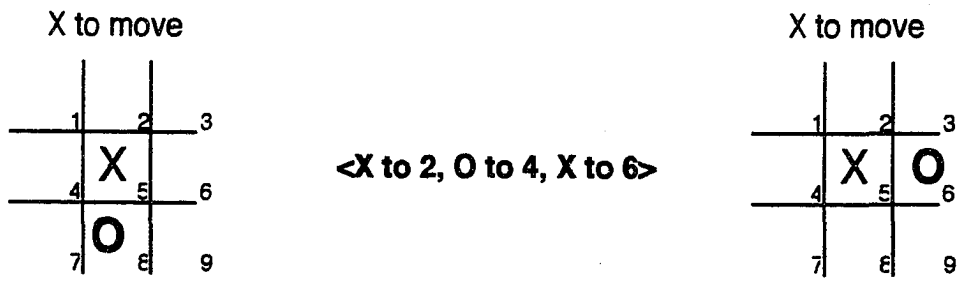


Figure 6: (a) A lose-tic-tac-toe state, (b) a sequence recorded in the SHT for this state, (c) a symmetrically equivalent state where the sequence is also applicable

2.2.2: Filtering the SHT Tables

TP-Rote periodically sweeps the SHT's for both players and identifies frequently seen sequences, such as Figure 6(b), through the counters, *state-seen* and *sequence-seen*. Sequences so identified are concatenated as appropriate, merged into a partial tree, and inserted into the filtered hash table (*FHT*). The index into an FHT is a state, such as Figure 5(a). The value retrieved from an FHT is a tree of actions that follows in subsequent decisions, such as Figure 5(b). Because only a small amount of information from an SHT is kept, and because the information is condensed into trees rather than sequences, an FHT is orders of magnitude smaller than the subsequently discardable SHT from which it is built. After an FHT is constructed for each agent, the SHT's are no longer needed.

Table 1 is a high level description of *Filter*, the selection process that builds an FHT. Each state in an SHT is considered in turn. If a state has occurred often enough (*state-seen* > *significant*), *Filter* calculates the *likelihood* of each sequence, *sequence-seen/state-seen*, which estimates the likelihood that a sequence will be executed again when the state or its symmetric equivalent recurs. *Filter* calls *Attempt-Build* on each *high-likelihood* sequence, (i.e., > *cutoff*), to use the sequence as a building block for a larger sequence whenever possible. *Attempt-Build* (also in Table 1) applies the sequence to the state, forming a new state with which it re-indexes into an SHT to find the new state's sequences and concatenate them to the previous sequence. So long as the next segment of the sequence also has a high likelihood, *Attempt-Build* is called recursively to build sequences of maximum possible length. Finally, all the constructed sequences for a particular state are merged into a tree with a dummy root *R*, and recorded in the FHT.

The dummy root simplifies the representation by allowing each FHT entry to remain a tree rather than a forest of trees.

Table 1: *High-level versions of the algorithms, Filter and Attempt-Build, that construct a filtered hash table from a sequence hash table. Filter loops through the sequence list of each significant state in the SHT, and for each high-likelihood sequence calls Attempt-Build to build up this sequence in as many ways as possible. Attempt-Build applies the sequence to the state, and determines the high-likelihood sequences of the new state formed. It attaches each of these new sequences to the original sequence to form a long-sequence and recursively calls itself with the original state and each long-sequence to continue building up this sequence as much as possible. The heuristic Do-Not-Maintain is described on page 41.*

```
Filter (SHT, FHT)
for each state in SHT do
   when used(state) = false and state-seen(state) > significant
        retrieve the state's sequence-list from the SHT
        long-sequences ← for each sequence in sequence-list
                              calculate likelihood of sequence     ;i.e., seq-seen/state-seen
                              if likelihood > cutoff
                                    collect Attempt-Build (state, sequence)
                         end for
        sequence tree ← Merge-into-Tree (long-sequences)
        insert (state, sequence tree) into FHT
        used(state) ← true     ; state should no longer be considered for an entry to FHT


Attempt-Build (state, sequence)
new-state ← apply sequence to state
used(new-state) ← Do-Not-Maintain(new-state)
when used(new-state) = false and state-seen(new-state) > significant
        retrieve the new state's sequence-list from the SHT
        long-sequences ← for each new-sequence in new-sequence-list
                              calculate likelihood of new-sequence
                              if likelihood > cutoff
                                    long-sequence ← attach(new-sequence, sequence)
                                    collect Attempt-Build(state, long-sequence)
                         end for
        return long-sequences
```

To illustrate the routines in Table 1, consider some state *A*, with *state-seen* = 75, sequence *j*, *sequence-seen* (j) = 50, and *cutoff* = .25. Since *j* has a likelihood of 2/3, Filter

calls Attempt-build, which executes *j* on state *A*, forming a new state *B*. Using *B*, Attempt-Build re-indexes into *SHT*, retrieves B's sequences, and again checks *B's state-seen* value and *sequence-seen* values to retain only the important ones. If the algorithm finds two such sequences, say *k* and *l*, then Attempt-Build concatenates *j* and *k* and is recursively called to continue the *jk* sequence. It also concatenates *j* and *l* and is recursively called to build that one too. Suppose, at the end, five sequences are formed: *j-k-m-n, j-k-m-o, j-k-p-r, j-l-s-t* and *j-l-w-x*. These five sequences are merged into a tree of moves, displayed in Figure 7, with *j* below the dummy root, and subtrees for *k* and *l*, and inserted with the original state *A* into the FHT.

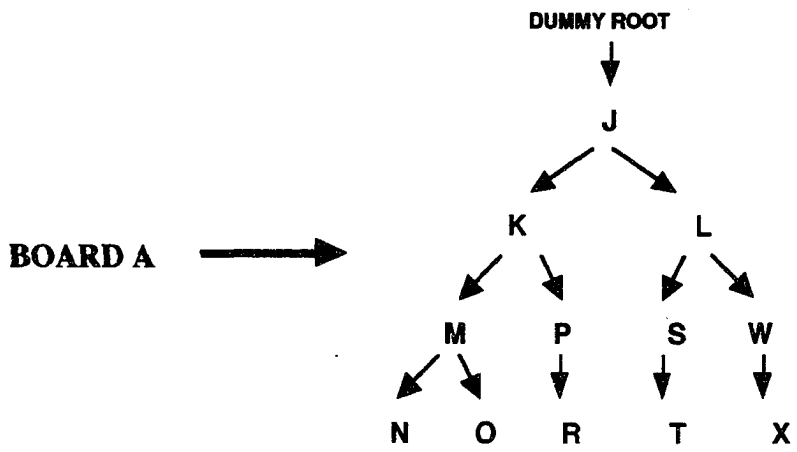


Figure 7: A sequence tree in FHT

With this method, any even-depth tree is accessible from the concatenation of even sequences, and any odd-depth tree is accessible from the concatenation of some even-depth sequences with an odd number of odd-depth ones. TP-Rote can be set to create odd trees (*FHT-odd*), that begin with a player's moves, or even trees (*FHT-even*), that begin with an opponent's moves, or both. If both types of trees are created, identical sequences found in both an odd tree and an even tree are eliminated from the even tree to conserve space in the FHT's.

The *Filter* and *Attempt-Build* algorithms use a cutoff to determine if a sequence should be inserted as part of a tree in the FHT. The cutoff is determined by the equation

$$cutoff = cutoff\text{-}base - (state\text{-}seen - significant)/\ significant * .01$$

where the hand-tuned *cutoff-base* is decremented by .01 for each additional (beyond *significant*) significant number of times the state occurred. By determining the cutoff as a function of the state's significance, and then matching this cutoff value against a sequence's likelihood, both the state's and the individual sequence's importance are considered by the filter, independently. As the frequency with which a state is seen rises, its cutoff drops, increasing the chances that the sequences played on this state will be stored in the FHT. This further refines the selective-retention filter (Markovitch and Scott 1993).

Filter and *Attempt-Build* select which states become keys in the FHT and which sequences are inserted as part of an FHT tree. A heuristic, *Do-Not-Maintain*, was developed to assess whether a state reachable via a sequence tree in the FHT also merits status as a key in the FHT. If such a state often arose through different paths, it remains eligible as a key in the FHT. To illustrate this heuristic, let us return to the example above that dealt with state *A*, sequence *j*, state *B* and sequences *k and l*. In the example, *Attempt-Build* built a tree of moves that followed state *A* in play and then inserted *A* with this tree into FHT. State *A* is then marked as *used* by *Attempt-Build*. The question remains whether the states reached via *A's* sequence tree in FHT, for example state *B*, should also be marked as *used* or not. It is possible that such states often arose through different paths, not necessarily following *j's* execution on state *A*. Such a state should be allowed

to become part of other trees in the FHT or even become a separate key in the FHT. In this way the alternate paths which led to B will not be ignored.

To accomplish this, an extra counter, *sum-seq-to-it* was added to each state in the SHT. While building the FHT, this slot is constantly updated to hold the total sum of all the *sequence-seen* values of the sequences that led to this state and are included in any FHT sequence tree. *Do-Not-Maintain* checks the difference between a state's *state-seen* value and its *sum-seq-to-it* value, which indicates how often a state arose not preceded by the paths already in the FHT. A difference greater than *sig* is an indication that this state merits status as a separate key in the FHT. Using our above example, *Do-Not-Maintain* is called on state *B* with sequence *j*. Since sequence *j* leads to state *B*, and is included in an FHT tree, *Do-Not-Maintain* updates *B's sum-seq-to-it* value by adding to it the *sequence-seen* value of *j*, 50. It then calculates the difference between *B's state-seen* value and *B's sum-seq-to- it* value. If the difference remains greater than *sig*, *Do-Not-Maintain* returns false and *B* merits status as a separate entry in the FHT. If, however, the difference is less than *sig*, *Do-Not-Maintain* returns true and *Attempt-Build* sets the *used* value of this state to true to indicate that this state should not be allowed to become a separate entry in the FHT. After the construction of an FHT is complete, a one-time *refilter* procedure loops through the FHT entries using this same check to determine if any entry should be removed from the FHT. It is possible that at the time an entry was added to the FHT it was appropriate, but after further entries were added to the FHT, it warrants removal.

2.2.3: Integrating TP-Rote

To exploit the FHT tables learned by TP-Rote, a new Advisor, *Multi-step,* was added to Hoyle's first tier. *Victory* (which makes a winning move) remains as the top advisor,

followed by *Enforcer* (which executes sequences of moves set into action) and then Multi-step. Recall that tier-1 Advisors are perfectly correct and are consulted in a pre-specified order. Multi-step is included in tier 1 with the expectation that the FHT sequences are correct. This will be the case in game playing if the SHT was collected only in won or drawn contests at a draw game with at least one error-free contestant. If the best available player is expert but imperfect, TP-Rote introduces some risk of error. Hoyle's second tier of heuristic Advisors was left unchanged for the TP-Rote experiments.

Each agent has an FHT-odd beginning with its own actions and an FHT-even beginning with the other agent's actions. Since sequences that begin with one's own actions involve less matching time, Multi-step checks FHT-odd first. If Multi-step can retrieve a sequence tree for the current state from FHT-odd, it skips over the dummy root R, selects and recommends one of the possible first moves in the tree, and passes the subtree of the selected move to Enforcer. If no sequence tree was retrieved from FHT-odd, Multi-step indexes into FHT-even with the state from which the other agent just moved. Multi-step checks any retrieved tree for the opponent's last action, recommends a continuation from the tree, and sends the subtree of the selected move to Enforcer. Enforcer continues recommending the next possible moves, so long as the opponent's moves follow the paths in the tree. Note that these decisions do not entail any extensive computation, pattern matching, or search. Multi-step and Enforcer make quick and easy decisions based on memorized sequences known to be frequently useful. Figure 5 can be used as an example to illustrate this process. Suppose Hoyle is playing X and the state in Figure 5(a) arises. Multi-step indexes with this state into FHT-odd and retrieves the tree

in Figure 5(b). It recommends a move to 9 and passes the subtree beneath 9 to Enforcer. At Hoyle's next turn, Enforcer checks whether the opponent's last move matches a next move in the tree, (in this case is a move to 6), and, if it matches, Enforcer mandates a move to 4.

If the opponent does not follow a sequence tree in use, the sequence tree is abandoned, and Multi-step uses the next state to re-index into the FHT in search of a new sequence tree. Planning methods often attempt to patch an old plan that did failed in an effort to save time replanning from scratch (Koęnig, Furcy et al. 2002) (Koenig and Likhachev 2002). TP-Rote, however, is not a method that creates plans; rather it learns and memorizes commonly used plans and reuses them. Since it does not plan, it has no ability to patch plans either. Instead, it checks if there is another stored plan that is applicable in this new state.

2.3: Experimental Design

Hoyle learns to play a game during *training*, where it plays and/or observes play in a tournament of contests. During training, Hoyle gathers game-specific knowledge and uses this knowledge in subsequent play. After training is finished, testing evaluates Hoyle's game-playing ability in tournaments against various contestants. No knowledge is acquired during testing. A *run* in Hoyle is training followed by testing. Because Hoyle is non-deterministic, breaking voting ties at random, each *experiment* reported here averaged data from ten runs.

There are various training environments under which Hoyle can learn to play. A *perfect agent* is an external program that plays a randomly-chosen best move in any state. An x-reasonable agent plays reasonably $x\%$ of the time, and perfectly $100–x\%$ of the

time. *Reasonably* means that a winning move one ply away is taken, a losing move two ply away is avoided, and otherwise a random, non-losing move is chosen. TP-Rote's experiments used *perfect-vs-increasing* training, a training environment where the learner observes increasingly skilled players compete against a perfect agent. In each run, Hoyle watched a random agent, a 70%-reasonable agent (i.e., a novice), a10%-reasonable agent (i.e., an expert) and a perfect agent compete, in that order, each in a 30-contest tournament against a perfect agent. Altogether, the total training consisted of 120 contests. The four watching tournaments were followed by four 50-contest testing tournaments (a total of 200 testing contests) that pitted Hoyle against the same three reasonable agents and a perfect agent.

To test TP-Rote, two experiments were done with each of the two games: the first with the standard version of Hoyle, and the second with TP-Rote as well. These experiments were done to measure the change in execution time during testing as the result of adding the TP-Rote method to Hoyle. In these experiments the parameter *cutoff-base* was set to .25. The parameters *significant* and *n* (in the cutoff function) were set to 20 for lose-tic-tac-toe, and to 5 for five men's morris. The games with the smaller search spaces received higher *significant* and *n* values so that the FHT's would not memorize too much of the space. This enables us to determine how much time can be saved by memorizing only a small fraction of the space that tends to be used often in real play.

A straightforward rote memorization of a table of complete state descriptions with their corresponding moves would save as much time as, or more than, TP-Rote. The advantage of TP-Rote's sequence tree representation is that it is able to save time within far less space than the full table approach. To measure how much space TP-Rote saves, a

third experiment was done. For this experiment, a new Advisor, *Imitate*, was developed and added to Hoyle instead of Multi-Step. Imitate is an Advisor that converts TP-Rote's FHT sequence trees into a *situation-action representation* (each situation paired with an action), instead of the *situation-sequence representation*. Imitate creates a hash table, where the key is a full state, and its value is an action, or list of possible actions to take. The hash table represents exactly the states reachable via all the FHT sequence trees. When it is Imitate's turn to move, it takes the current state, indexes into the hash table and, if the state is found, returns an action to take. An action is randomly selected if Imitate finds more than one possibility in the hash table. In the third experiment, Hoyle used Imitate instead of Multi-step. All other parameter settings remained the same as the other experiments. Imitate may be as fast or faster than TP-Rote, but, it is an intractable approach for larger game boards. It was used only to reveal how much space TP-Rote's compact representation saves.

2.4: Results

All results reported in this thesis are statistically significant at the 95% confidence level. The graphs in Figures 8(a) and 8(b) show execution times during testing for the three experiments, for lose tic-tac-toe and five men's morris, respectively. In both games, the results show execution time speedup with TP-Rote over the standard Hoyle, for all opponents but the random player when playing five men's morris. For lose-tic-tac-toe, Figure 8(a), although speedup occurred when testing against all opponents, there was a much more dramatic improvement when testing against the two better players. After scaling up to five men's morris, the same relationship is found in Figure 8(b); that is, TP-Rote offers much more speedup over standard Hoyle when playing against the two better

opponents. This seems to indicate that better players are more sequence-oriented in their play, enabling TP-Rote to make more decisions. This concept will be further discussed in Section 2.5. On the lower end of the spectrum, TP-Rote offers no speedup at all in five men's morris when competing against a random player. Because a random player plays 100% randomly, following no sequences at all, TP-Rote is of little use. The reason that random did offer speedup for lose tic tac toe is because the state space for that game is so small, that even a random player played some sequences that were found in the FHT tables. Once the method was applied to the significantly larger space of five men's morris, however, the true impact of the method becomes clearer. Five men's morris has an average contest length of 40, and a game graph of approximately 7 million states, over 1,000 times as large as lose tic-tac-toe's. Thus it can only be said that TP-Rote improves execution time performance against better players.

Table 2 describes the sequence trees learned by TP-Rote. The standard deviations show that the trees learned for each of the ten runs were similar, even though each run had a different training experience. The small number of trees learned, combined with the similarity of these trees, indicate that important segments of the search space are being found again and again. Figures 9(a) and 9(b) illustrate the number of decisions TP-Rote (i.e. Multi-step and Enforcer) made during testing against the various opponents. TP-Rote was able to make many more decisions when playing against the two better opponents, that is, the more often the opponent repeated sequences, the more often Hoyle was able to utilize TP-Rote's sequence trees. More TP-Rote decisions lead to greater speedup.

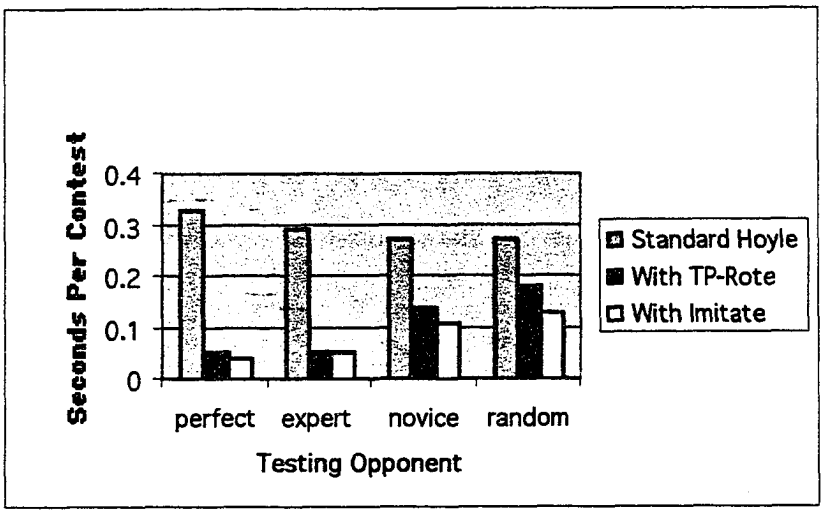


Figure 8(a): Execution Speed for lose tic-tac-toe

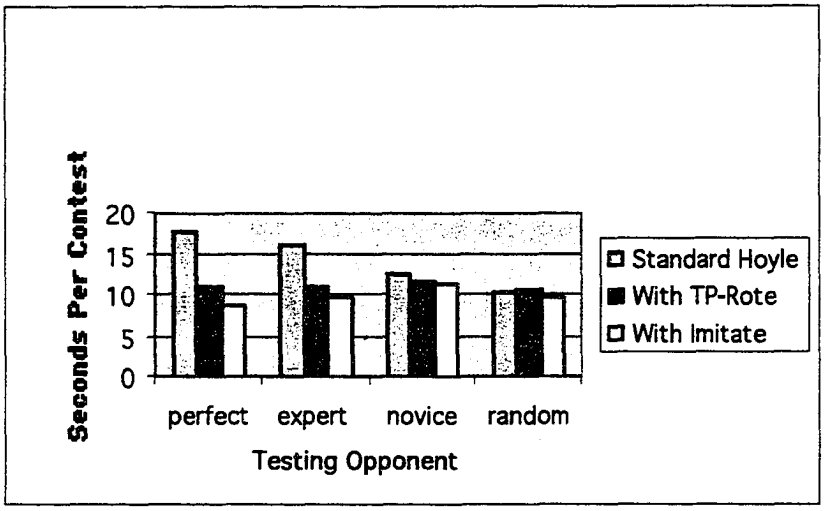


Figure 8(b): Execution Speed for five men's morris
(drawn to different scale than Figure 8(a))

Table 2: Number of trees learned by TP-Rote and number of sequences incorporated in these trees, averaged over 10 runs

	# trees	stnd. deviation	# sequences	stnd. deviation
Lose Tic-Tac-Toe	4.10	0.62	12.90	2.34
Five Men's Morris	69.00	2.74	210.90	19.45

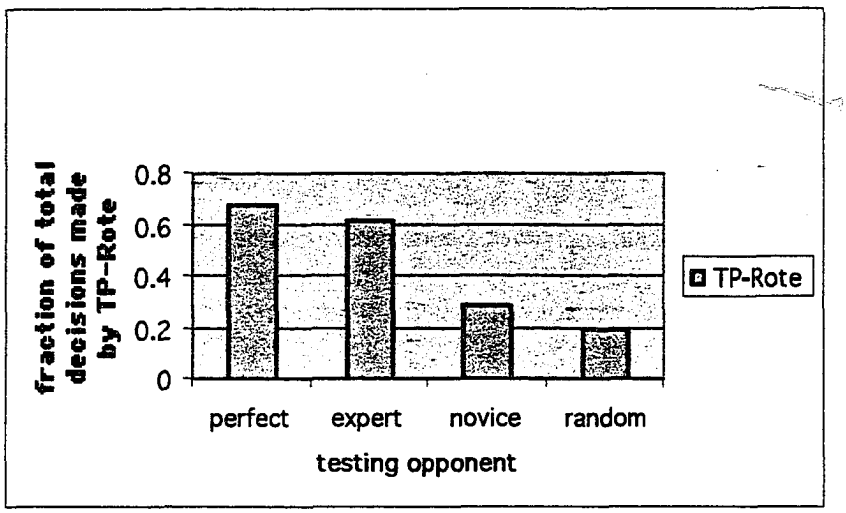


Figure 9(a): TP-Rote usage for lose tic-tac-toe

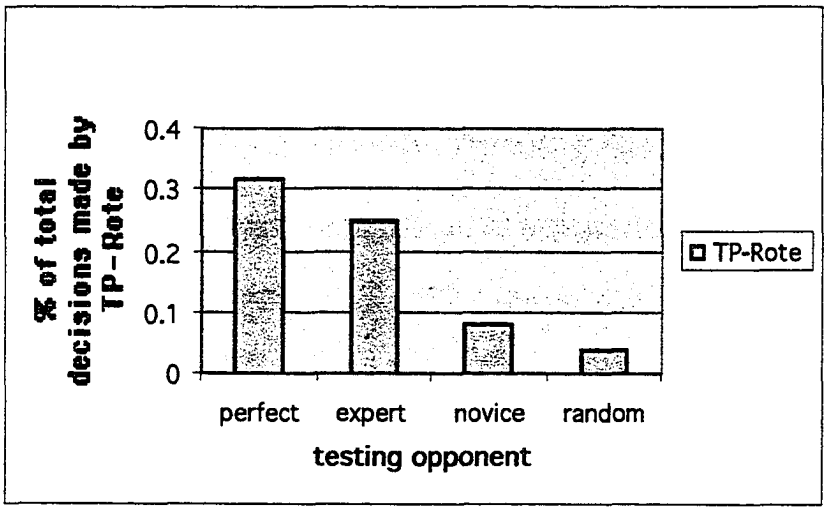


Figure 9(b): TP-Rote usage for five men's morris
(drawn to different scale than Figure 9(a))

Returning to Figures 8(a) and 8(b), one can see that Imitate led to at least as much speedup as TP-Rote, against all opponents at all levels, both in lose tic-tac-toe and five men's morris. Figures 10(a) and 10(b) compare the number of decisions Imitate was able to make, with the number of decisions TP-Rote was able to make. Again, Imitate was able to make at least as many decisions as TP-Rote, against all opponent-levels, both in

lose tic-tac-toe and five men's morris. Interestingly, Imitate was used significantly more than TP-Rote for lose tic-tac-toe when playing against the two worse opponents, and for five men's morris when playing against the two better opponents. Imitate's greater utilization in these cases can be explained as follows. Although Imitate's hash table represents exactly those states reachable via TP-Rote sequence trees, the hash table can be indexed into directly to retrieve a decision for any state found in it, no matter which sequence produced the state during real play. TP-Rote sequence trees, on the other hand, can offer a decision only if real play follows a sequence that was memorized within a sequence tree. Considering this, it is quite striking that even in the cases where Imitate is used significantly more, the usage increase is not as dramatic as one would expect. This indicates that there do indeed exist portions of the game tree (i.e., sequence trees), that are important, and that states within these sequence trees will usually arise in these same paths.

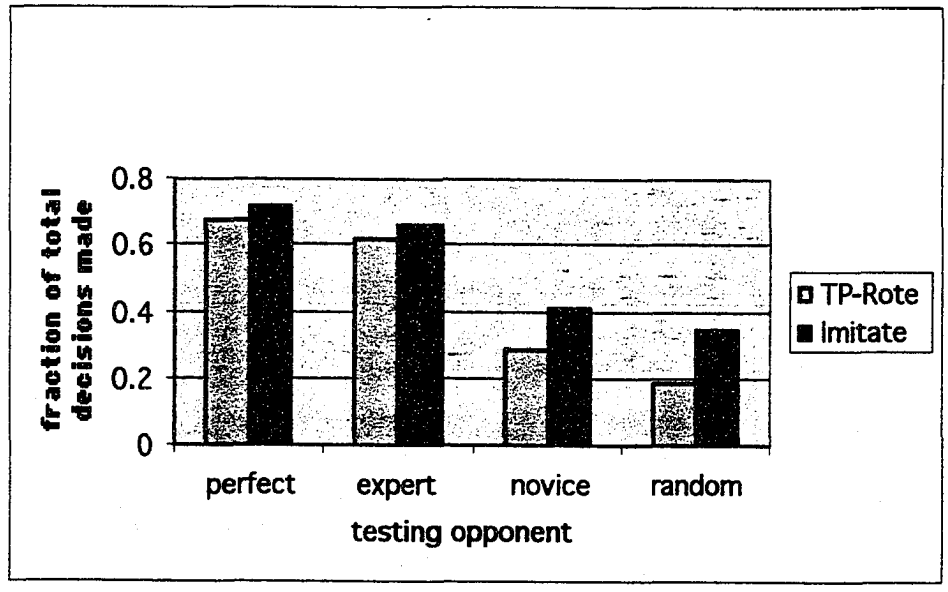


Figure 10(a): Comparing TP-Rote usage and Imitate usage for lose tic-tac-toe

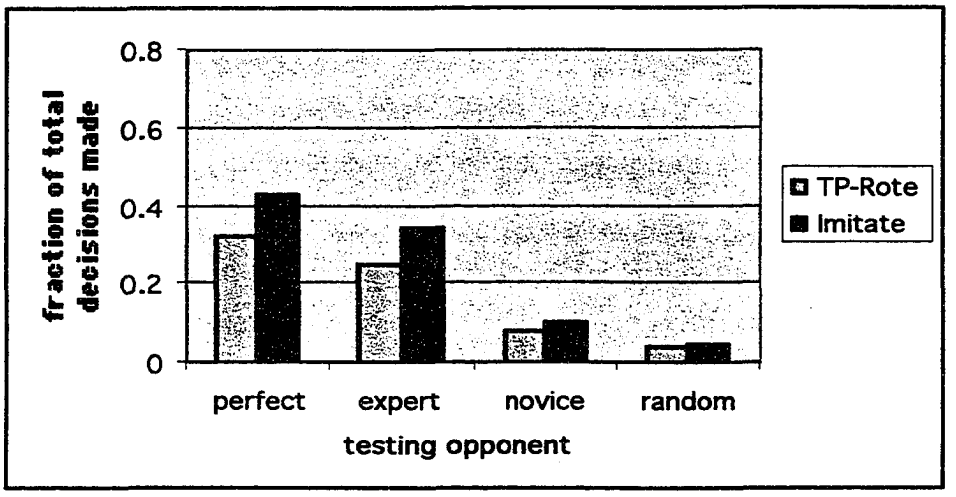


Figure 10(b): Comparing TP-Rote usage and Imitate usage for five men's morris

Table 3: Situation-sequence space (TP-Rote) vs. situation-action space (Imitate) after 120 training contests (space needed for states and moves stored)

	#states	st. dev.	# moves	st. dev.	space needed	st. dev.
Lose Tic Tac Toe						
TP-Rote	4.10	0.62	42.40	6.84	79.10	11.45
Imitate	14.80	1.72	17.80	2.32	151.00	17.69
Five Mens Morris						
TP-Rote	68.50	2.74	697.40	82.35	1793.40	190.44
Imitate	206.90	17.56	255.20	23.33	4076.30	349.36

Recall that Imitate was developed to prove that, although this simple approach might lead to as great execution time speedup and usage as TP-Rote, its space requirements would be far greater. Table 3 compares the space requirements for states and moves stored by Imitate and TP-Rote. While TP-Rote stores many more moves than Imitate, it stores far fewer states. Since states are much larger than moves, this leads to an impressive space saving of 48% for lose tic-tac-toe and 56% for five men's morris. Space on the chart is measured using one unit of measurement for every board position, one unit

for a lose tic-tac-toe move, and three units for a five men's morris move, since a five men's morris move involves the slide-from, slide-to, and capture positions. The larger the game board as compared to the size of a move, the greater the space saving will be.

Table 4 records Hoyle's performance during testing for the three experiments. It measures performance by *reliability*, the percentage of wins and draws that Hoyle achieved in each of the testing tournaments, and *power*, the percentage of wins that Hoyle achieved. The goal of TP-Rote was to improve decision-speed, not performance. TP-Rote can only make decisions that were already known to the standard Hoyle, so there is no reason for it to change the performance level. The purpose of Table 4 is to document that TP-Rote did not cause any statistically significant adverse affect on Hoyle's performance.

Table 4: Performance data for lose tic-tac-toe and five mens morris after 120 training contests

	Perfect	Expert		Novice		Random	
	Reliability	**Reliability**	**Power**	**Reliability**	**Power**	**Reliability**	**Power**
Lose tic tac toe							
Standard Hoyle	99.60	97.80	15.20	96.60	62.20	97.80	74.00
With TP-Rote	99.60	99.80	16.20	98.00	63.60	97.00	74.20
With Imitate	99.60	98.80	14.40	97.00	58.20	97.40	73.00
Five men's morris							
Standard Hoyle	84.40	78.40	14.60	87.60	67.40	95.80	85.20
With TP-Rote	83.40	83.80	16.00	90.80	69.60	98.20	87.00
With Imitate	79.60	81.20	15.00	91.80	67.20	99.00	88.40

2.5: Discussion

TP-Rote creates a compact temporal pattern memorizer. Experiments with TP-Rote indicate a significant speedup in Hoyle's play, simulating the gradual shift to "play without thought" observable in human game players, particularly during openings and endgames. Examination of the contents of the FHT's shows that TP-Rote learned opening

and endgame sequences as well as others. Results show that the time spent in learning the sequences is repaid later in time saved when using them. TP-Rote's limitation is its inability to generalize beyond symmetry. In the proper setting it learns perfectly correct sequences, but it limits sequence usage to states that have been seen before. TP-Rote accomplished its two-fold purpose: to save time without requiring too much space. Although the SHT's can grow as large as half the maximum number of states seen during experience (in the unlikely case that every state seen for a player is significant and is stored), they are discarded after the condensed FHT's are completed. If necessary in larger spaces, entries in the SHT with low state-seen values could be intermittently discarded, to reduce the SHT size. Results show that the FHT's require dramatically less space than a hash table of state-move pairs.

An interesting side result was that sequence usage is correlated with level of expertise. The higher the level of expertise, the more the appropriate play followed sequences in decision-making rather than making isolated decisions. This lends credence to the theory that better players tend to function in small areas of the search space (Epstein 1995). It also fits with the cognitive experiments of Chase and Simon, mentioned in Chapter 1, which provided evidence that better chess players have some long term memory structure of move-sequences which they use to aid in their choice of moves (Chase and Simon 1973). Hence, any method developed to capitalize on sequences will be more useful when playing against better opponents.

TP-Rote was initially developed for lose-tic-tac-toe but scaled well to the much larger space of five men's morris. This method should be applicable to larger domains as well. Because TP-Rote's simple algorithm only considers the limited number of states

seen during experience, and does not search or reason, it could be applied to any size domain. Its success in speeding up decision making, however, depends on how many stereotyped sequences a domain possesses, and how often they recur, since this is what TP-Rote is geared to learn.

Chapter 3: TP-Context

This chapter examines TP-Context, a learning method that learns action sequences for abstracted state descriptions, rather than for complete state descriptions. Section 3.1 gives an overview of the TP-Context learning algorithm, while Section 3.2 details the representation and implementation. The experimental design is outlined in Section 3.3, followed by the experimental results in Section 3.4. The chapter concludes with a discussion in Section 3.5.

3.1:The Algorithm

Learned sequences associated with complete state descriptions, as in TP-Rote, cannot be used in novel situations. The goal of the second method, TP-Context *(Temporal Patterns through Context),* is to find the appropriate context for a move sequence, that is, the portion of the board that motivated the sequence. Such generalizations cover many states and enable more widespread use of the sequences learned. This is especially important for games with large state spaces, where identical situations may not often re-arise.

To discover context, TP-Context exploits the knowledge inherent in the sequences actually experienced in game-playing contests. Each sequence seen in a contest is associated with the set of states upon which it was executed. The underlying assumption is that there is a commonality among these boards that led to the same set of moves. The discovery problem is thus narrowed to a small cluster of states from which the context can be extracted. Once the context is extracted, it is associated with the sequence it triggers as a <context, sequence> pair. TP-Context builds up a store of Sequence Advisors, as <context, sequence> pairs which are used to play the game. If an Advisor determines that its context matches the current state, it suggests the matching sequence as

a course of action. Because the Sequence Advisors are learned inductively, a weight is learned for each Advisor during subsequent play, according to how much its advice simulates perfect play. In the end, only the highest weighted Advisors are retained.

3.2: Representation and Implementation

3.2.1: Building the Context Hash Table

As in TP-Rote, TP-Context examines the actions in a sequential experience for sequences of pre-specified lengths. To associate sequences with states, TP-Context organizes the data collected in a *CHT* (Context Hash Table), where a key is a normalized sequence and its stored value is a list of the states where the sequence began. TP-Context indexes into the CHT with each extracted sequence, and places on the sequence's value list the state at which the sequence's execution began. Figure 11, for example, shows the lose tic-tac-toe sequence <X to 1, O to 3, X to 7> that began at six different states. The sequence in Figure 11 is the key into the CHT, while the list of states is its stored value. The full state descriptions become the candidate set from which the context is induced.

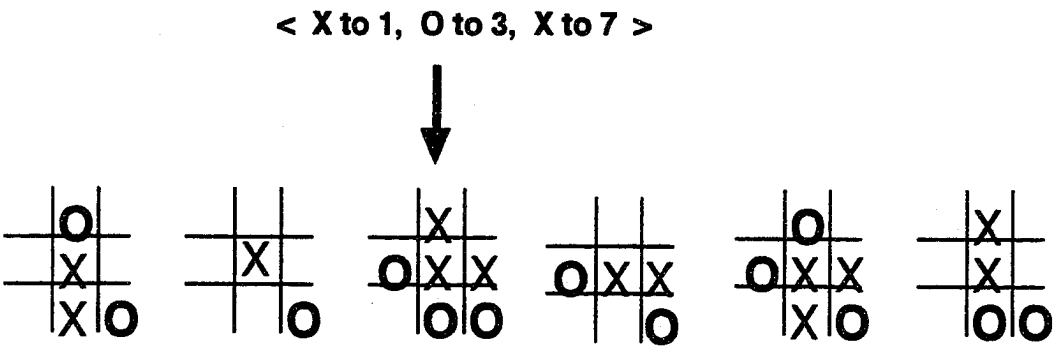


Figure 11: The knowledge stored in the CHT

Some domains have *solos*, where an agent at a certain point in a sequential experience executes a few actions that belong together, no matter what the other agent does. If TP-Context gathered only sequences that included both agents' actions, the other agent's varied responses could cause each instance of the same solo to appear to be a different sequence, and TP-Context would store each one as a separate key in the CHT. Therefore, TP-Context also gathers sequences of actions that belong to one agent only. This way, solos are stored under one index in the CHT.

3.2.2: Extracting the Context

Once a sequence is associated with only a small set of states, context discovery becomes more tractable. In the current implementation for game playing, TP-Context identifies the maximal common pattern that exists in all the states as the context for the sequence. TP-Context takes each location of the board as a separate feature and examines its value on each board, retaining only features with the same value on all the boards. Any location whose value is not retained is labeled # for "don't care." The result is the maximal common pattern. Figure 12, for example, is the context the algorithm finds for the lose tic-tac-toe sequence <X to 1, O to 3, X to 7>.

The locations underlying the sequence are 1, 3, 5, 7, and 9 containing blank, blank, X, blank, and O respectively. Rather than associate the sequence with all six states, TP-Context associates the sequence with the pattern found on the generalized board in Figure 12. If this pattern matches a state during future play, <X to 1, O to 3, X to 7> can be suggested as a possible course of action, whether or not the state is among the six originally seen.

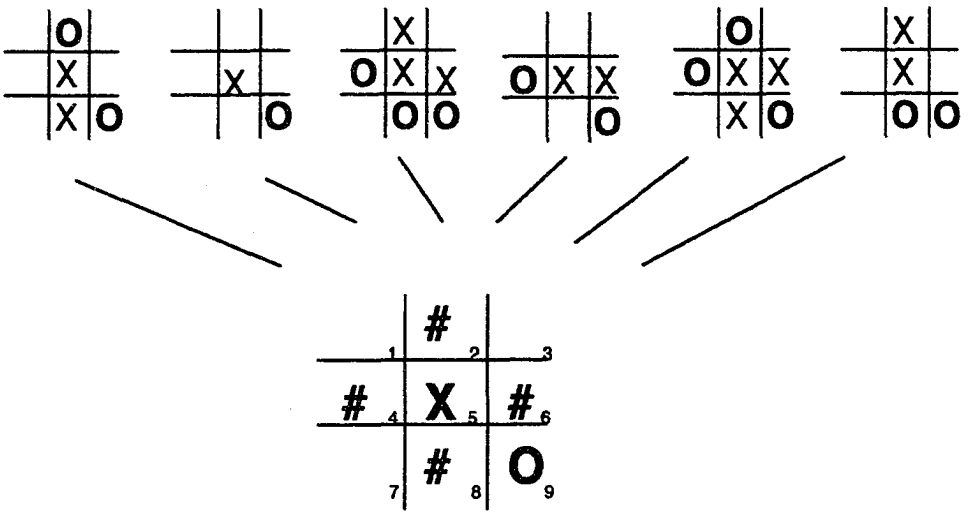
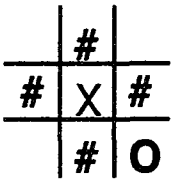


Figure 12: The extracted context

3.2.3: Creating Context Advice

After every *k* experiences, TP-Context sweeps through the CHT, examining one sequence at a time. To ensure the usefulness of the learned sequences, TP-Context, like TP-Rote, only induces the context for a sequence that occurs often. For such a sequence, it extracts the maximal common pattern as described above. Next, it proposes a *context pair* of the form <*context, sequence*>, where *context* is a set of conditions (the maximal common pattern) that suggests the appropriateness of *sequence*'s execution The context pair in Figure 13, for example, was learned for X in lose tic-tac-toe. Contexts and their associated sequences are normalized for the symmetries of the two-dimensional plane, and are applicable anytime the identical context or one of its symmetric variations arises. With each *k* experiences, the candidate sets for each sequence grow larger, and new, more general context pairs are created from the CHT. If more than one sequence leads to the formation of the same context, the sequences are merged into a tree of actions and only one context pair is created.



, < X to 1, O to 3, X to 7 >

Figure 13: (a) Context and (b) it's associated sequence for X in lose tic-tac-toe

Given their inductive origin, these pairs are not used immediately in problem solving; care is taken to filter out the correct pairs from the incorrect ones. TP-Context uses a variant of *PWL* (Probabilistic Weight Learning), a perceptron-like algorithm, to weight them. The algorithm determines how well a context pair simulates expert behavior After every experience, it takes the states where it was the expert's turn to act, checks whether a context pair's advice supports or opposes the expert's decision, and revises the pair's weight accordingly. A pair's weight rises consistently only when it is reliable and frequently applicable. There are, to be sure, states that match Figure 13(a) where 1 would not be the correct move. If there were many such states, Figure 13(a) would have a low weight.

3.2.4: Choosing the Representation

TP-Context's context-pairs are similar to Patsy's pattern Advisors, although they are created via different methods. The two major differences are that Patsy's pattern Advisors are <context, move> pairs rather than <context, sequence> pairs, and that the generalized contexts of the Patsy Advisors are limited in shape by built-in templates. To check if an Advisor's move was correct, Patsy used the following method. First it took the current state and created all symmetrically equivalent states. Then it pattern-matched

the generalized context to each of these symmetrically equivalent states to check if the context was found in any possible orientation. If it did, the Advisor was weighted according to how well the move it recommended simulated an expert's play. Although this method worked, the pattern matching process required a lot of computation time.

Because TP-Context Advisors are not limited in shape by templates, TP-Context spawns a much large number of potential context-pairs than Patsy does before deciding which ones to retain. Matching each pair's context to the current state in order to determine and weight its move would entail a lot of time using the standard pattern matching comparison process. This was not an issue in TP-Rote, because complete state descriptions allow the straightforward use of a hash table, eliminating the need for pattern matching. An original representation and method was developed for TP-Context that enabled it to use hash tables even though it uses generalized contexts, which greatly reduced the pattern matching time. Furthermore, a pre-processing stage incorporated each context in the hash table along with all its symmetric equivalents and their associated sequences. In this way, TP-Context avoids Patsy's need to recreate all symmetric equivalents at each decision point.

The underlying idea in developing TP-Context's representation was to determine a unique index for each context pair, representing the pair's context, to be used as a key in a hash table. The value of the key would be the pair's sequence. Update-Hash-Table, in Table 5, describes this process. Once the hash tables are created, given any current state and a context, the context's index should be able to be reconstructed correctly only if the context is found on the current state. Using this index, one can then retrieve the

appropriate sequence from the hash table. Check-Hash-Table, in Table 5, details how this is done.

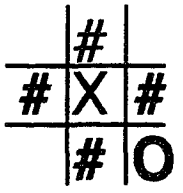
Table 5: *High-level versions of the algorithms, Update-Hash-Table that constructs the Context Pair Hash Table and Check-Hash-Table that uses the Context Pair Hash Table. Create-All-Equivalents is described on page 8.*

```
Update-Hash-Table(context-pair, context-location-list)
    all-orientations ← Create-All-Equivalents(context-pair, context-location-list)
    for each triple in all-orientations do
        context ← first(triple)
        sequence ← second(triple)
        con-loc-list ← third(triple)
        string1 ← null
        for each location in con-loc-list do
            string2 ← concatenate(location, value of this location on context)
            string1 ← concatenate(string1, string2)
        end for
        index ← concatenate(string1, idnum)
        insert (index, sequence) into Context Pair Hash Table
        save con-loc-list on all-context-location-lists with original context-pair
    end for


Check-Hash-Table(state, context, all-context-location-lists)
plans ← for each context-location-list in all-context-location-lists do
        string1 ← null
        for each location in context-location-list do
            string2 ← concatenate(location, value of this location on state)
            string1 ← concatenate(string1, string2)
        end for
        index ← concatenate(string1, idnum)
        sequence ← lookup(index, Context Pair Hash Table)
        collect sequence
    end for
return plans
```

Recall that each learned context pair is stored in normalized from. Before Update-Hash-Table is called, a learned context pair is associated with a unique identification number, and a *context-location-list*, the list of the locations on the board that consist of

the context (i.e., those locations that do not have a don't care value). For example, the context-location-list for the context pair in Figure 13 is (1 3 5 7 9). Update-Hash-Table is then called for each context pair. The first thing Update-Hash-Table does is call Create-All-Equivalents to compute all symmetric orientations of the context, each one matched with the sequence and the context-location-list in the correct symmetric form for that orientation. Create-All-Equivalents returns a list of triples (context, sequence, context-location-list), including all orientations of the context, even the original normalized form.



(1 3 5 7 9) **'1nil3nil5X7nil9O'**

(a) **(b)** **(c)**

Figure 14: (a) A sample context (b) its context-location-list (c) the string formed by Update-Hash-Table using Figures 14(a) and 14(b)

'1nil3nil5X7nil9Oidnum' → **< X to 1, O to 3, X to 7 >**

(a) **(b)**

Figure 15: A Context Pair Hash Table entry formed by Update-Hash-Table (a) the key (b) its value

For each possible orientation, Update-Hash-Table loops through its context-location-list, and concatenates the location number and the value of this location on the context, forming a string 'location-number location-value location-number location-value...'. For example, the string formed for the context in Figure 14(a), using its context-location-list in Figure 14(b), is '1nil3nil5X7nil9O'; see Figure 14(c). The context-pair's

identification number is then concatenated to the end of the string creating a unique index for this context pair. The index is inserted as a key in the Context Pair Hash Table with its corresponding sequence as its value. Each variation of the context-location-list is then saved with the original normalized context pair for using the hash tables later on. Figures 15(a) and (b) show the entry in the Context Pair Hash Table for the context in Figure 14(a). Since an index is created for all orientations, if there are seven symmetric equivalents for one normalized context, there will be eight entries in the hash table. Because hash tables use indexing, adding entries to the hash table does not require much more time when using them.

Check-Hash-Table, in Table 5, accepts as parameters a current state, a context pair in normalized form, and a list of all the variations of the context's context-location-lists that were stored with it in the Update-Hash-Table routine. The job of Check-Hash-Table is to determine if this context exists on this current state, in any orientation, and if it does, to retrieve the appropriate sequence from the hash table. Check-Hash-Table forms an index for each of the context's context-location-lists, as follows: For each location in context-location-list, concatenate location number and the value of this location on the current state, to form a string 'location-number location-value location-number location-value'. (Note that Check-Hash-Table looks at the value of this location on the current state rather than on the context as Update-Hash-Table did.) Check-Hash-Table then concatenates the context pair's identification number to the end of the string formed and indexes into the Context Pair Hash Table. If the context indeed exists on this state in any orientation, one of the context-location-lists will recreate the correct index, which will be found in the hash table and used to retrieve the sequence that the context pair recommends. In case the

context exists in more than one orientation, all retrieved sequences are collected and returned as a list of possible plans.

To illustrate Check-Hash-Table with an example, consider the state in Figure 16. Suppose this is the current state and the program would like to know if Figure 14(a)'s context exists on this state. Patsy's method would first compute the seven symmetric variants of the state in Figure 16, and then match the context in Figure 14(a) to each of these states. Instead, Check-Hash-Table loops through all of Figure 14(a)'s context-location-lists and creates an index for each of them. In this case, the context is indeed found in its original normalized form, so the context-location- list (1 3 5 7 9) leads to the creation of the index 1nil3nil5X7nil9Oidnum which is found in the Context Pair Hash Table, as seen in Figure 15.

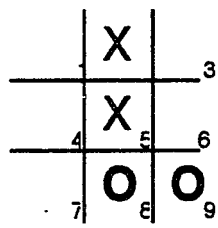


Figure 16: A sample current state in play

3.3: Experimental Design

3.3.1: The Design

The goal of TP-Context is to determine how much one can learn about a game from observing sequences played during experience. Specifically, we would like to discover how much knowledge can be gleaned from these sequences, generalized as <context, sequence> pairs and used to play the game. Certainly not everything about a game can be learned from sequences. However, how important a knowledge source this can be for learning programs is the open question that this method addresses.

There is an underlying difference between Hoyle and TP-Context. Hoyle uses pre-selected features known to be important for game-playing. These handcoded features, called Advisors, use learned knowledge, but they are not themselves learned. TP-Context's Advisors, on the other hand, are learned features based only upon played contests, with no other knowledge given to them. To test TP-Context, a new Hoyle-like player, *SeqLearner*, was created. This new player, described in the next section, does not use Hoyle's pre-specified Advisors since doing so would mask TP-Context's true strengths and weaknesses. It would not be clear what knowledge was learned by TP-Context and what was not. Furthermore, this approach has an advantage for other games and domains where it is unclear as to what the set of predetermined features should be.

Version I of SeqLearner, used in the first set of experiments was of a two-tier, Hoyle-like hierarchical framework. Tier 1 consisted of a single Advisor, Enforcer, similar to Hoyle's Enforcer, whose job is to continue a sequence that is already set into motion. Tier 2 consisted of a group of Sequence Advisors, (i.e., <context, sequence> pairs) with weights above a pre-set *threshold*. During play, control begins in tier 1, where Enforcer suggests a next move in an ongoing sequence. If no sequence is in the process of being executed, control passes to tier 2. Using the process discussed in Section 3.2.4, each Sequence Advisor compares its context with the current state; if it is applicable, it recommends the first action in its sequence. As in Hoyle's tier 2, a vote is taken in tier 2 to determine which of the actions suggested by all the Sequence Advisors should be chosen. Thus, actions that begin more than one sequence and actions whose Advisors have higher weights are more likely to be selected. If a sequence's first action is chosen, the rest of the sequence is sent to Enforcer, which merges the individual sequences it

receives into one sequence tree. As the contest continues, Enforcer recommends a next action from the branch in the sequence tree that the opponent followed. If no move is recommended in tier 2, a random move is chosen. The collective voice of the tier 2 Advisors can be viewed as a *planner*, which chooses a plan to follow, based upon the current context.

A more sophisticated version of SeqLearner, Version 2, shown in Figure 17, was used for the second set of experiments. Tier 1 remained the same as in Version 1, yet was followed by two more tiers. Tier 2 consisted of a three-segment hierarchy; the top segment included Advisors with weights ranging from .9 to 1.0, the middle segment included Advisors with weights from .8 to .89, and the bottom segment included Advisors with weights from .7 to .79. Each of the three segments in tier 2 function as a planner, in exactly the same way as Version 1's tier 2 did; however, they are consulted in order of the hierarchy as long as a plan is not yet selected. When consulted, a planner compares all its Sequence Advisor with the current state, to find the appropriate plans, and suggest the first action in these plans. The planner takes a vote to determine which of the actions suggested by all the Sequence Advisors should be chosen. If a sequence's first action is chosen, the rest of the sequence is sent to Enforcer, which merges the individual sequences it receives into one sequence tree. Enforcer continues the plan set into action as long as the opponent follows suit.

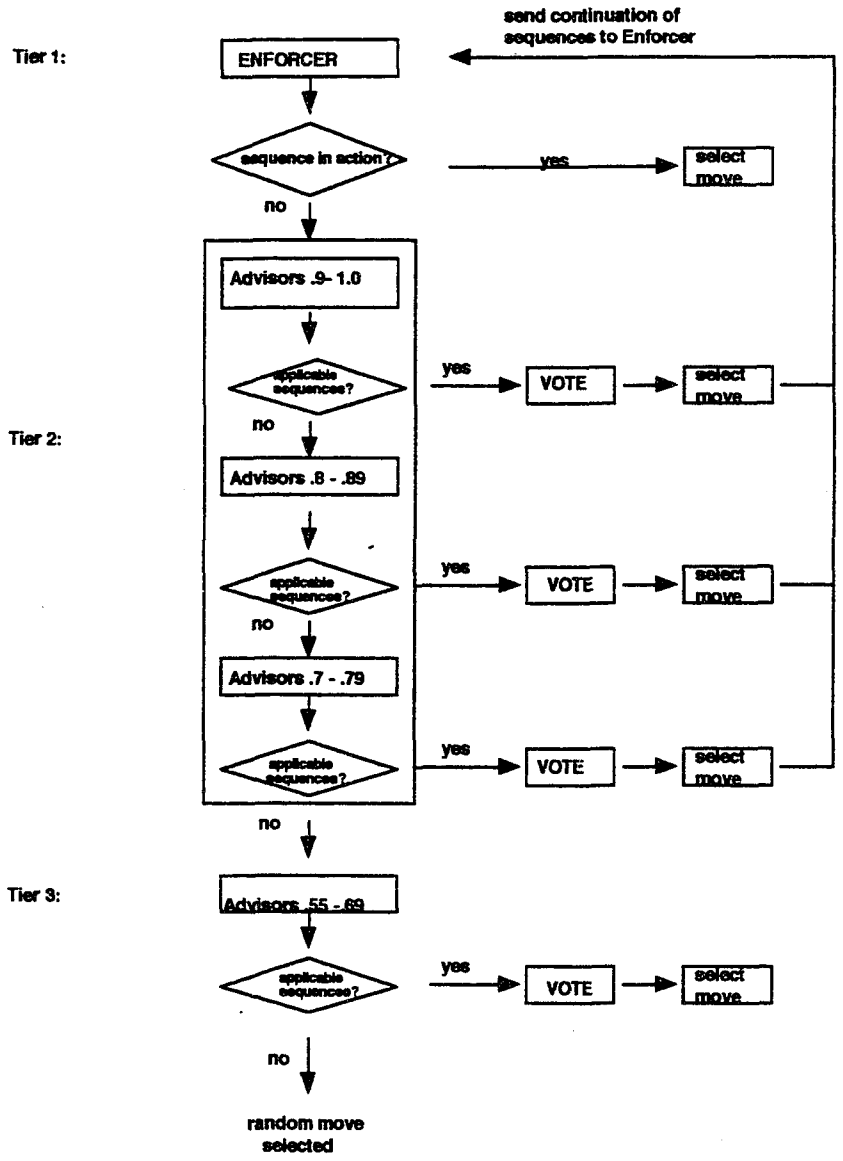


Figure 17: Version 2 of SeqLearner used in TP-Context's Experiment II

Tier 3 consisted of Advisors whose weights ranged from .55 to.69, and functions as a *pseudoplanner*. Like the tier 2 planners, the tier 3 Advisors find the applicable plans and select an action to take based upon them, but tier 3 does not send the continuation of the chosen plans to Enforcer. Tier 3 was allowed to give advice of one move, since advice from low weight Advisors is better than choosing a random move. However,

rather than considering the rest of the plans suggested by these low weight Advisors, the program rechecks the higher weight Advisors on the next move, in anticipation that a better decision will be made, and a more correct sequence will be set in action. Control passes from tier 1 through tier 3; if a move is still not selected then a random move is chosen.

Segmenting tier 2 and separating the low-weight Advisors as tier 3 serves multiple purposes. First, better advice is given even stronger preference than it was before. Second, by considering only one segment at a time, and stopping when advice is given, the program no longer has to consult the full set of Advisors on each turn, thereby decreasing execution time. Third, tier 3 was not given the privilege of using Enforcer. Partitioning the Advisors into separate tiers allows the program to make different rules for the better and worse Advisors.

3.3.2: The Experiments

First, a baseline experiment was conducted both for lose tic-tac-toe and five men's morris for purposes of comparison. (The games are discussed in Section 1.6 and the rules for these games can be found in Appendix A.) In the baseline experiments a random player competed against four different level opponents: a perfect player, an expert, a novice and a random player. Next, to evaluate TP-Context, two sets of experiments were conducted on lose tic-tac-toe and five men's morris. Each set consisted of one experiment on lose tic-tac-toe and one experiment on five men's morris, for a total of four experiments. The first set of experiments, used Version 1 of SeqLearner with a threshold of .55 (i.e., where <context sequence> pairs with weights above .55 became Sequence Advisors in tier 2, and those with weights below .55 were discarded). The second set of experiments used

Version 2 of SeqLearner, where the Sequence Advisors were partitioned and segmented according to their weight. Because SeqLearner is non-deterministic, breaking voting ties at random, each experiment reported here averaged data over twenty runs for lose-tic-tac-toe, and over ten runs for five men's morris. Each run consisted of a set of training contests where SeqLearner used the TP-Context method to learn as much as it could about playing the game, followed by a set of testing contests that evaluated SeqLearner's game-playing ability in tournaments against various contestants.

Like Hoyle, SeqLearner can learn to play in various training environments. All four of the TP-Context experiments used a variation of the *perfect-vs-increasing* training environment, an environment where the learner observes increasingly skilled players compete against a perfect agent. In this variation, termed *double-cycle perfect vs increasing*, SeqLearner observes increasingly skilled players, but after observing the best player, returns to watch the worst player and again repeats the cycle of observing increasingly skilled players. In each run, SeqLearner watched a random agent, then a 70%-reasonable agent (i.e., a novice), then a 10%-reasonable agent (i.e., an expert) and then a perfect agent compete, in a tournament against a perfect agent. It then repeated this observation for another four watching tournaments. The double cycle was necessary to allow the weights to develop correctly. During this training stage, SeqLearner used the TP-Context method to learn <context sequence> pairs and weight them accordingly. While training, the set of Sequence Advisors expands and contracts as various <context, sequence> pairs gain and lose their status as a Sequence Advisors. In each run, the eight watching tournaments (i.e., the training) were followed by four 50-contest testing tournaments that pitted SeqLearner against the same three reasonable agents and a perfect

agent. During testing, any partitioning of the Sequence Advisors in tiers 2 and 3 remain static, since weights no longer change.

Some parameters were different for lose tic-tac-toe and five men's morris, to accommodate the difference in these two games' search space. For lose tic-tac-toe, a much easier game to learn, the watching tournaments were of 20 contests each, whereas for five men's morris they were of 40 contests each. In both cases, the TP-Context algorithm learned new Advisors after each watching tournament was over, but for lose tic-tac toe, length 3 sequences were learned, while for five men's morris length 3 and length 5 sequences were learned, because five men's morris' contests run longer. Although TP-Context can learn any length sequence, the lengths chosen for the final experiments were hand-tuned.

3.4: Results

Table 6 displays the performance results of the baseline experiment and the two experiments done with both lose tic-tac-toe and five men's morris. Reliability is the number of wins + draws, while power is the number of wins. Results that represent statistically improved performance over the previous experiment are highlighted in bold, with standard deviation values below the reliability and power values in parentheses. Recall that TP-Context begins without any built-in knowledge, either of game-playing in general or of the particular game itself. It uses only the knowledge it learns from watching sequences in actual play. In light of that one does not expect TP-Context alone (i.e., without other learning methods) to lead to superior play. However, performance results are necessary first, to prove that much knowledge about a game is contained in

experienced sequences, and second, to measure TP-Context's ability to capitalize on this knowledge.

For lose tic-tac-toe, results of Experiment I (using Sequence Advisors with weights above .55) show that SeqLearner was able to win or draw between 66% and 70% of the time against the stronger players and between 83% and 85% of the time against the weaker players. Experiment I leads to significant improvement in both reliability and power over Baseline for all four opponents. A separate, unnumbered experiment was run with Version 1 of SeqLearner and a higher threshold than .55, to check if consulting only more highly-weighted Advisors would lead to better play. Interestingly, for lose tic-tac-toe this led to considerably worse play. This could be explained by the fact that when SeqLearner reaches a state where no Advisor gives advice, a random move is chosen. Hence, although fewer, more highly weighted Advisors may make better decisions when they are applicable, fewer Advisors also lead to a greater number of random decisions. Experiment II was an attempt to incorporate the advantages of both a higher and a lower threshold. On the one hand, higher weighted Advisors should be given priority in making decisions, on the other hand, when higher weighted advisors have no advice, lower weighted Advisors should give advice rather than SeqLearner resorting to random decisions. In Experiment II, all Advisors above .55 were partitioned into tier 2 and tier 3, and tier 2 was segmented, creating a hierarchical SeqLearner. This simple change enabled SeqLearner to win and draw the game of lose tic-tac-toe between 82 and 87% of the time against all opponents, significantly improving reliability against the perfect and expert players.

Table 6: Performance data for lose tic-tac-toe after 160 contests and for five mens morris after 320 contests (statistical improvements are highlighted in bold with standard deviations in parentheses below performance values)

	Perfect	Expert		Novice		Random	
	Reliability	**Reliability**	**Power**	**Reliability**	**Power**	**Reliability**	**Power**
Lose tic tac toe							
Baseline	20.50	25.80	9.00	49.20	37.70	58.40	46.10
	(4.14)	(5.33)	(4.12)	(7.57)	(6.67)	(7.28)	(7.84)
Experiment I	**66.00**	**70.80**	**12.80**	**83.50**	**42.70**	**85.30**	**55.40**
	(20.96)	(16.22)	(4.71)	(6.38)	(8.08)	(4.79)	(5.52)
Experiment II	**82.00**	**83.80**	12.10	85.10	46.70	87.00	57.40
	(13.24)	(9.73)	(4.22)	(5.08)	(7.08)	(4.17)	(7.02)
Five men's morris							
Baseline	0.20	1.60	1.20	22.40	14.20	54.00	36.00
	(0.60)	(1.20)	(0.98)	(4.27)	(5.90)	(7.27)	(6.81)
Experiment I	**39.60**	**34.20**	1.00	**28.80**	9.20	54.20	17.40
	(13.11)	(11.29)	(1.34)	(4.66)	(2.23)	(7.18)	(6.39)
Experiment II	**57.80**	**45.00**	0.80	**45.00**	**29.00**	**73.00**	**51.20**
	(13.22)	(10.05)	(1.33)	(4.58)	(5.81)	(8.35)	(7.44)

Five men's morris has a significantly larger space than lose tic-tac-toe; it gives a picture of how this method scales. After running Experiment I, SeqLearner learns a fair amount of knowledge about how to play the game, significantly improving reliability over Baseline for the perfect, expert, and novice players. Interestingly, Experiment I actually decreased the power of the novice and random players. Not only does basing moves upon sequence knowledge alone not give a player enough power to win, it may actually give too little variation for the worse players which follow few, if any sequences. After running Experiment II (TP-Context with levels) with five men's morris, performance results show improved reliability, over Experiment I, for all four opponents and improved power for the novice and random players. Unlike Experiment I, Experiment II also improved the power of the worse players compared to Baseline. Furthermore, SeqLearner was able to win and draw against a perfect player 57.80% of the time. (It performs a little worse against the mid-level opponents.) The results indicate that using hierarchical segments and tiers is a good idea.

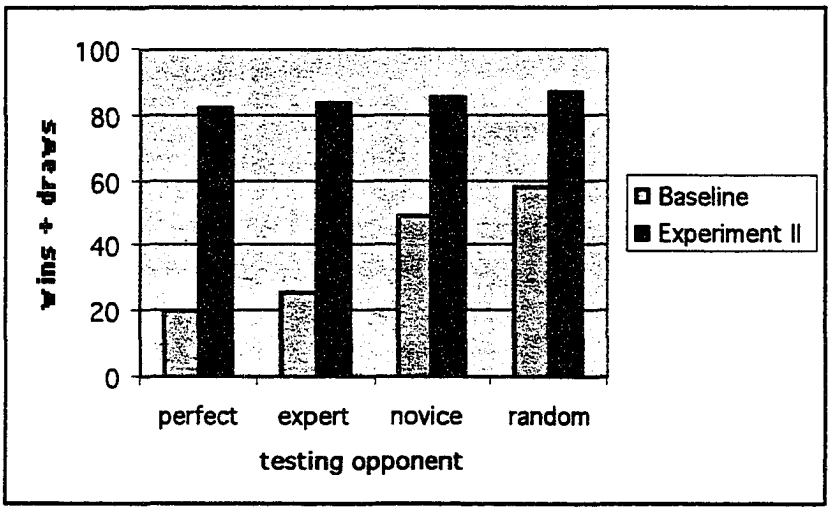


Figure 18(a): Baseline's reliability compared with TP-Context Experiment II's reliability for lose tic-tac-toe.

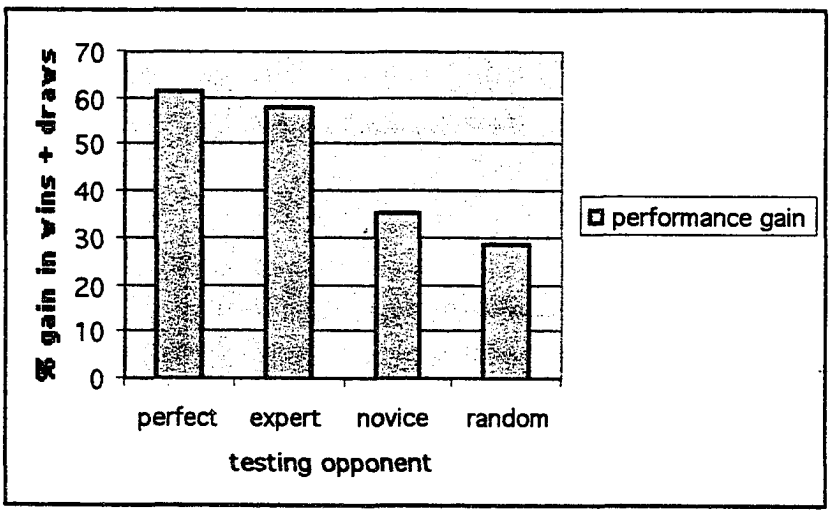


Figure 18(b): Performance Gain of TP-Context for lose tic-tac-toe

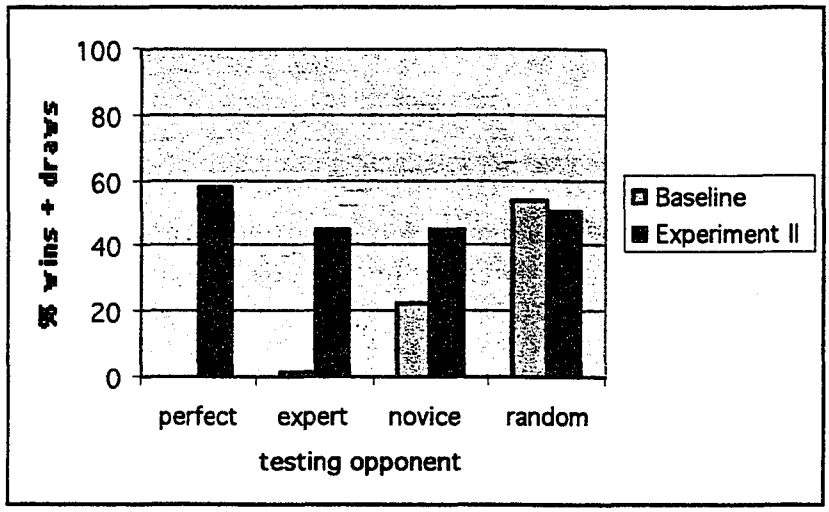


Figure 18(c): Baseline's reliability compared with TP-Context's (Exp II) reliability for five men's morris.

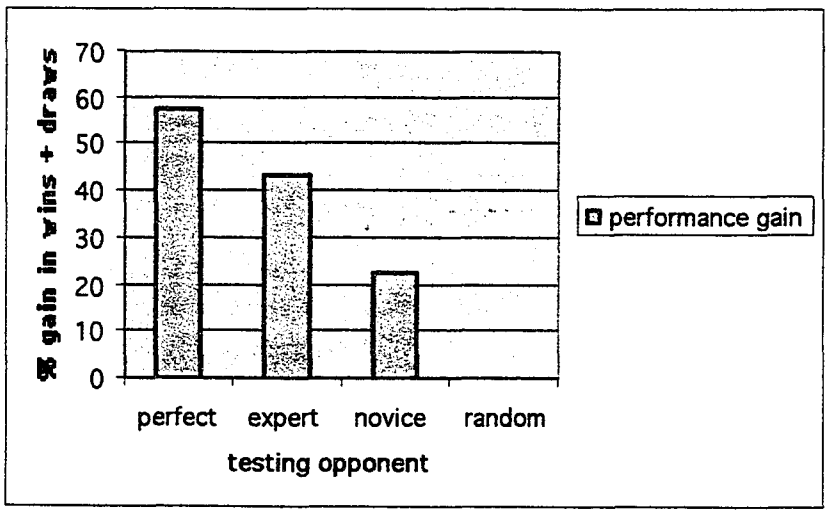


Figure 18(d): Performance Gain of TP-Context for five men's morris

Figures 18(a) –(d) summarize the results of these experiments. For lose tic-tac-toe, Figure 18(a) compares the reliability of Baseline, (i.e., a random player against the four level opponents), with the reliability of SeqLearner after running Experiment II,

against these four opponents. The performance gained by TP-Context in reliability, from Baseline to Experiment II, is graphed in Figure 18(b). For five men's morris, these graphs are found in Figure 18(c) and 18(d). Clearly, TP-Context is most useful for learning techniques against better opponents. As with TP-Rote, the better opponents rely more on sequence knowledge. Machine learning's great challenge is learning knowledge that is neither explicit nor easy to build in by hand. The better the opponent, the more one has to rely on learning programs rather than hand-built programs. Hence an algorithm like TP-Context, that performs better against stronger opponents, is an important step in game learning, and is a prospective component for multiple-method learning programs.

Table 7: Percentage of Context Pairs retained as Sequence Advisors

	#Context Pairs	std. dev.	# Advisors	std. dev.	%retained
Lose Tic Tac Toe					
Exp I: (Advisors >.55)	36.80	4.02	31.55	3.25	85.73%
Exp II: (with hierarchy)	38.05	4.80	33.05	4.30	86.85%
Five Mens Morris					
Exp I: (Advisors > .55)	3838.50	82.25	1599.30	33.90	41.66%
Exp II: (with hierarchy)	3843.80	117.84	1615.60	77.77	42.03%

It is also important to examine the amount of space and time required by TP-Context. Table 7 displays the number of <context, sequence> pairs spawned by the method, and the percentage retained as Sequence Advisors after weighting the pairs in play. In both Experiments I and II, the pair spawning and weighting processes remain identical, so the numbers are about the same. For lose tic-tac-toe the vast majority of context pairs spawned achieve weights greater than .55 and are therefore maintained as Sequence Advisors. This indicates that lose tic-tac-toe is a highly sequence-oriented

game. For five men's morris, approximately 40% of the context pairs spawned are weighted high enough to become Sequence Advisors.

Table 8: Breakdown of learned Sequence Advisors in Exp II by tiers and segments

	Lose Tic Tac Toe	std. dev.	Five Mens Morris	std. dev.
Tier 2				
Segment 1 (weight .9 - 1.0)	17.80	2.79	575.50	48.79
Segment 2 (weight .8 - .89)	3.80	1.54	210.40	8.44
Segment 3 (weight .7 - .79)	6.65	1.84	277.90	20.97
Tier 3 (weight .55 - .69)	3.20	1.44	496.30	20.79

Table 8 further examines the quality of the Advisors learned in Experiment II, by their weights. Interestingly, both in lose tic-tac-toe and five men's morris, the group of Advisors with weights .9-1.0 forms the largest segment of tier 2, larger than the combination of tier 2's segments 2 and 3 and tier 3. This means that many of the Advisors retained give excellent advice. In five men's morris, the second largest group is in tier 3. This indicates that partitioning will decrease execution time, since very often tier 3 is not reached in the decision-making process.

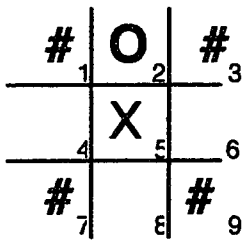
Table 9 reports the average Sequence Advisor usage (i.e., the percentage of states during testing where SeqLearner had enough knowledge to make a non-random decision) and the execution time (in seconds) that SeqLearner required per contest during testing. (The machine and OS used in these experiments are described in Appendix B.) For lose-tic-tac-toe there is no difference in Advisor usage between Experiments I and II, while for five men's morris there is a slight, yet significant decrease in Advisor usage with Experiment II. Although both Experiments I and II retain the same Advisors (those above .55), in Experiment II the Advisors with weights .55 -.69 are only given the ability to

suggest the first move of their sequence. Since lose tic-tac-toe's tier 3 in Experiment II was very small there was no decrease in Advisor usage; however, since five men's morris' tier 3 in Experiment II formed the second largest group of Advisors, there was a slight decrease in Advisor usage with an improved performance.

For the small space of lose tic-tac-toe, no difference is found in execution time between the two experiments. This is because roughly the same number of Advisors was retained in both experiments, and more than half of them were found in the top segment of tier 2 in Experiment II. However, moving to the larger space of five men's morris shows us the true difference in execution time between the two experiments. Experiment II has significantly lower execution time that Experiment I because it partitioned the Advisors into tier 2 and tier 3, and further segmented tier 2. Thus, the results indicate that Experiment II is the better approach.

Table 9: Average Sequence Advisor usage and execution time during testing

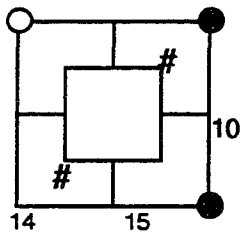
	Advisor usage	std. dev.	execution time	stn.dev.
Lose Tic Tac Toe				
Exp I: (Advisors > .55)	0.82	0.06	0.02	0.00
Exp II: (with hierarchy)	0.81	0.07	0.02	0.00
Five Mens Morris				
Exp I: (Advisors > .55)	0.92	0.01	3.24	0.25
Exp II: (with hierarchy)	0.89	0.01	2.55	0.20



< X to 8, O to 6, X to 4 >

(a) **(b)**

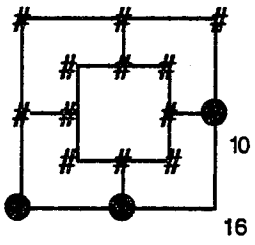
Figure 19: (a) A Sequence Advisor learned for Player X in lose tic-tac-toe (b) the sequence it suggests for X



< W to 10, B to 14, W to 15 >

(a) **(b)**

Figure 20: (a) A placing stage Sequence Advisor learned for Player White in five men's morris (b) the sequence it suggests for White



< B slide from 10 to 16
W anywhere
B slide from 16 to 10 >

(a) **(b)**

Figure 21: (a) A solo sliding stage Sequence Advisor for Player Black in five men's morris that learned to create a mill (slide from position10 to position16) and shuttle in and out of the mill (b) the sequence it suggests for Player Black

Examination of the actual Advisors learned show that TP-Context learns concepts that are important for each game. Figure 19 gives an example of a Sequence Advisor

learned for lose tic-tac-toe that achieved a perfect weight of 1.00 after 160 training contests. Figure 20 is a Sequence Advisor learned in the placing stage of five men's morris. Figure 21 shows a solo sliding Sequence Advisor which learned to create a mill and shuttle in and out of the mill once it is achieved, which is a crucial sequence for winning five men's morris. (An identical Sequence Advisor was also learned for Player White, with White and Black markers reversed.) The two five men's morris Advisors in Figures 20 and 21 achieved weights of 1.00 after 320 training contests. All Sequence Advisors learned are applicable elsewhere on the board, when a symmetrically equivalent context arises.

3.5: Discussion

TP-Context learns action sequences for generalized states. By examining actual, experienced sequences, TP-Context discovers the context underlying a sequence. Although this allows response to novel situations with a learned sequence, overcoming TP-Rote's limitation, TP-Context's sequences lack the guaranteed correctness of TP-Rote's. Results show that the TP-Context method, beginning with no knowledge, can learn a lot about playing the simple game of lose tic-tac-toe, and scales to the much more difficult game of five men's morris. Surprisingly, after using the TP-Context method, SeqLearner can draw against a perfect five men's morris player more than 50% of the time. Results also show that the most dramatic improvement gained from applying the TP-Context method occurs when testing against the better players. This correlates both with the results of the TP-Rote method, and with Chase and Simon's results that better chess players have move-sequence structures stored in long-term memory, which they use in play.

Rather than consider all possible legal sequences, TP-Context induces contexts from the limited number of sequences seen during experience. Again, this factor should enable it to scale to larger spaces. However, for the induction to be correct in a large domain, many training examples will be needed. Therefore, for games with many more moves per contest and much larger boards, the TP-Context approach may require some adaptation. Instead of collecting all sequences, additional game-specific knowledge could specify when sequences should be collected. For example, one can define locality for a game, and collect only sequences that fall in the same locality. In Go, for example, sequences of moves in the same geographic vicinity are usually related; those would be the ones to learn. The algorithm that looks for the underlying context would also concentrate on a vicinity instead of the entire board. In a game like chess, however, locality would be defined both by proximity to the locations moved to, and by the squares those moved markers attack and defend.

Although the algorithm in TP-Context for the maximal common pattern is efficient, it does not consider the value of more than one position as a time, which may well be important to the actual context. Hence the algorithm may over-generalize, that is, drop locations whose values were not identical, but whose alternatives for values were important. TP-Context handles this problem by filtering the contexts learned during further play and maintaining only those that appear to be correct over time. To eliminate the problem entirely, the feature representation for learning contexts would have to be *complete*, able to express any possible combination of the locations on the board as a feature. Completeness comes at the expense of efficiency, however.

Although the TP-Context method has proved its ability to learn a lot about a game, on its own it cannot support perfect play. There is much knowledge about a game that cannot be captured with its limited view. In a real world domain, TP-Context would do best if combined with clear, well-defined knowledge already known about the domain. TP-Context could then learn more about the domain from actual experience and, together with the built-in knowledge, perform in a knowledgeable fashion.

Chapter 4: TP-Sitact

This chapter discusses TP-Sitact, an extension of the TP-Context method. Section 4.1 explains the motivation for TP-Sitact and gives an overview of the TP-Sitact algorithm. Section 4.2 details the representation and implementation. The experimental design is outlined in Section 4.3, followed by results in Section 4.4. The chapter concludes with a discussion in Section 4.5.

4.1:The Algorithm

After completing TP-Context a question arose. Would situation-action Advisors perform just as well as situation-sequence Advisors? Situation-action Advisors would advise one move based on a situation's context, rather than a sequence of moves as situation-sequence Advisors do. Just as Imitate shifted TP-Rote's situation-sequence representation to a situation-action representation, TP-Sitact explores this representational shift with TP-Context. The goal was to determine what effect such a representational shift would have on space, time and performance.

TP-Sitact begins by creating <context, sequence> pairs using the TP-Context algorithm. Rather than use these pairs to create Sequence Advisors, it converts each <context, sequence> pair into a set of <context, action> pairs that corresponds to the original <context, sequence> pair. TP-Sitact weights these <context, action> pairs in further play, according to how much their advice simulates a perfect player, and retains those with weights above a certain threshold as *Sitact Advisors.* A Sitact Advisor determines if its context matches the current state, and if it does, suggests its matching action as the next move. After learning, these Sitact Advisors are tested to determine how

well they perform. The algorithms for shifting TP-Context's representation are outlined in Figure 10 and discussed in detail in the following section.

Table 10: *High-level versions of the algorithms, Create-Sitact-Advisors and Execute-Move-on-Pattern that convert a Sequence Advisor to its corresponding Sitact Advisors*

```
Create-Sitact-Advisors(context con-locations sequence)
  Create-New-Advisor(context con-locations (first sequence))
  new-contexts ← (list (context con-locations))
  for i from 0 to (sequence-length – 3) by 2 do
        move1 ← ith move in sequence
        move2 ← ith + 1 move in sequence
        next-move ← ith + 2 move in sequence

        new-contexts ← for each pair in new-contexts do
              append (execute-move-on-pattern move1 pair)
        new-contexts ← for each pair in new-contexts do
              append (execute-move-on-pattern move2 pair)
        Sitact Advisors ← for each pair in new-contexts do
              if legal-pattern-move(next-move pair)
                    append Create-New-Advisor(pair next-move)
  return Sitact-Advisors

Execute-Move-on-Pattern(pair move)
  context ← first(pair)
  con-locations ← second(pair))
  new-contexts ←
        if not abstracted(move) do
              new-context ← apply move to context
              new-con-loc ← Revise-Con-Loc(move con-locations)
              return list of (new-context new-con-locations)

        if abstracted(move) do
              legal-moves-within-pattern ← legal-pattern-moves(context)
              context-pairs←
              (loop for move in legal-moves-within-pattern do
                    if move is (move to #) or (slide # to #)
                          collect (context con-locations)
                          else new-context ← apply move to context
                              new-con-loc ← Revise-Con-Loc(move con-locations)
                              collect (new-context new-con-loc))
  return new-contexts
```

4.2: Representation and Implementation

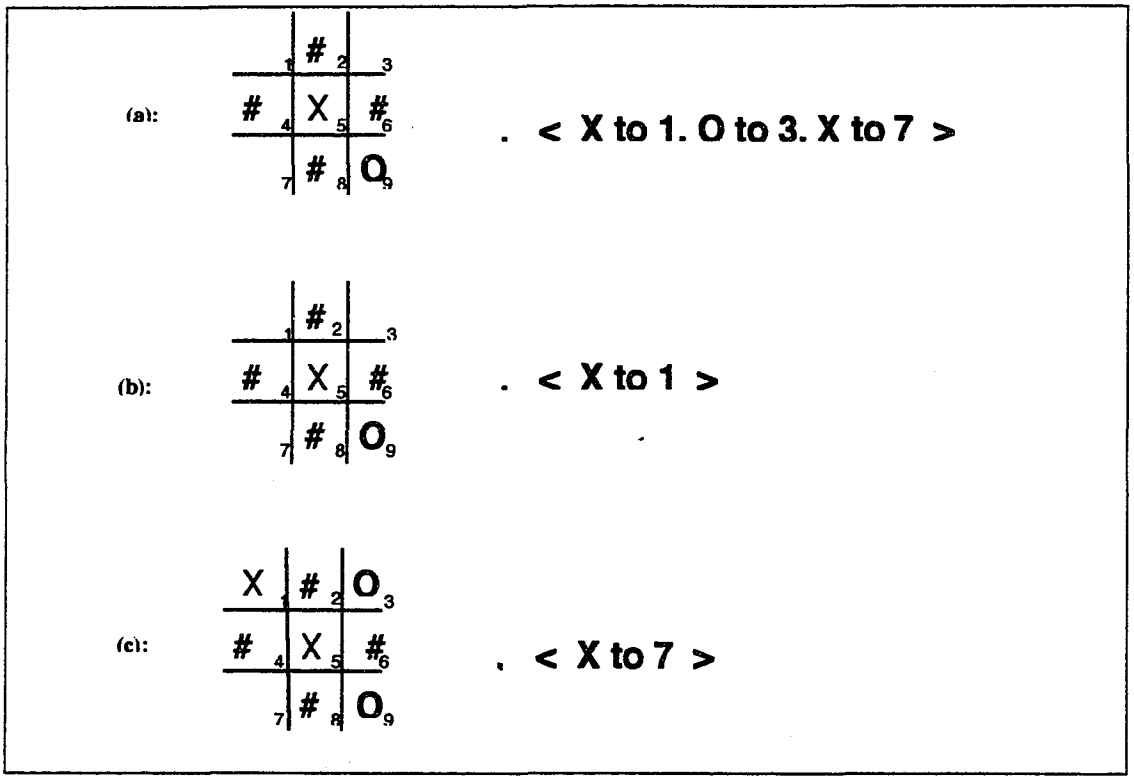


Figure 22: (a) A lose tic-tac-toe Sequence Advisor learned for Player X
(b) and (c) its two corresponding Sitact Advisors

The representational shift from Sequence Advisors to Sitact Advisors is outlined in the algorithms *Create-Sitact-Advisors* and *Execute-Move-on-Pattern* in Table 10. The following four cases, displayed in Figures 22 through 25, demonstrate how these algorithms work. Case 1, in Figure 22, is a simple lose tic-tac-toe example. Figure 22(a) is a Sequence Advisor learned for Player X in lose tic-tac-toe; its corresponding Sitact Advisors, created by Create-Sitact-Advisors, are found in Figure 22(b) and 22(c). First, Create-Sitact-Advisors creates a Sitact Advisor using the original Sequence Advisor's context, paired with the first move in the original sequence (shown in 22(b)). The algorithm then loops down the sequence, executing a player's move and its opponent's

response on the context and creates a new Sitact Advisor. The new Sitact Advisor consists of the new context formed after these two moves are applied, paired with the move in the sequence that follows these two moves. Here, one such new Advisor is created, after <X to 1> and <O to 3> are applied, and is displayed in 22(c). Had the original sequence been longer, the algorithm would repeat this process, forming more Sitact Advisors, until the end of the sequence. Since the original Sequence Advisor in Figure 22(a) was learned for Player X, all the corresponding Sitact Advisors formed are also learned from X's perspective.

The representational shift is slightly more complicated when dealing with solo Sequence Advisors. Recall that a solo Advisor contains a one-player plan in which the opponent's moves are abstracted out of the sequence. Hence, in order to form the corresponding Sitact Advisors all possible opponents' moves on the context must be accounted for. For lose tic-tac-toe, a game with only placing moves, the opponent can move to any empty location or to any location which contains a # (since it is potentially empty). Case 2, in Figure 23 contains a solo example for lose tic-tac-toe. Figure 23(a) is the original solo Sequence Advisor learned for Player X in lose tic-tac-toe, while Figures 23(b)- 23(e) show its corresponding Sitact Advisors after Create-Sitact-Advisors is called.

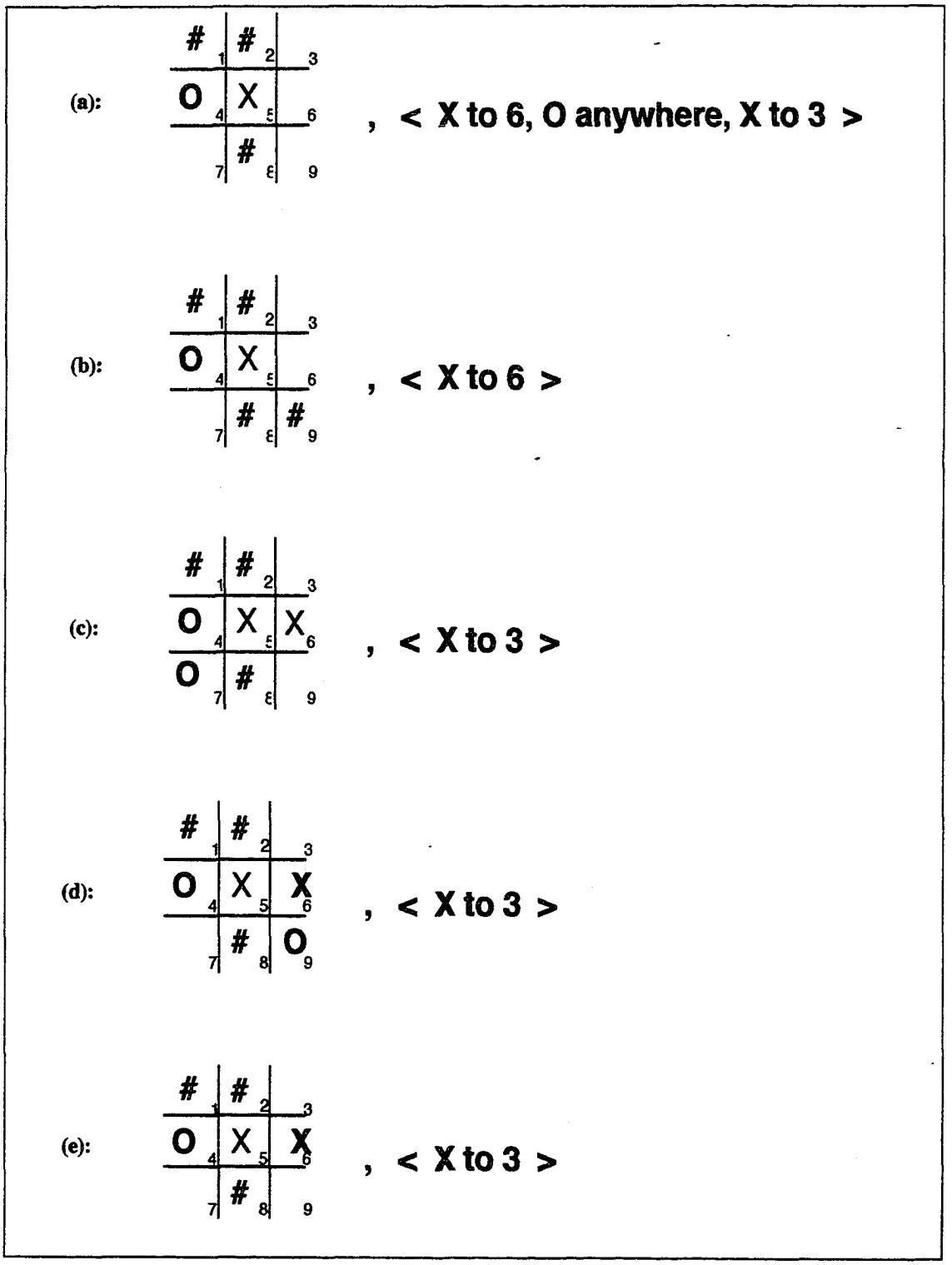


Figure 23: (a) A lose tic-tac-toe Solo Sequence Advisor learned for Player X (b) a Sitact Advisor formed using the original context pointing to first move in the original sequence (c) – (e) three Sitact Advisors formed after X's first move and all O's possible subsequent moves are executed

The first Sitact Advisor (in Figure 23(b)) corresponds to the original context, paired with the first move of the original sequence. Afterwards X's move to 6 is executed, and all possible moves to O are executed to form all potential new contexts. Execute-Move-on-Pattern checks if a move is abstracted or not; if it is not, it executes it on the context, and revises the context's locations to reflect the locations occupied by the new context. If the move is an abstracted move, it calls *legal-moves-within-pattern* to generate all possibilities for the abstracted move. Here O's possible moves on Figure 23(a)'s context are to 3, 7 or 9 which are empty, or to 1, 2 or 8 which hold #. The move to 3 is eliminated, since once O moves to 3, the context formed cannot suggest a move to 3, which is the next move in the sequence. Figures 23(c) and 23(d) contain Advisors formed after executing X's move to 6, and then O's move to 7 and 9 respectively. Figure 23(d) is a Sitact Advisor created to incorporate O's possible move to #. If O moves to a # location, this does not affect the context since the location remains abstracted as #. So the context used in Figure 23(d) is the context after X's move to 6 is applied (disregarding O's move) and paired with a move to 3, the next move in the sequence.

Case 3, in Figure 24, is a simple five men's morris example (i.e., without solo moves). Figure 24(a) shows the original five men's morris sliding Sequence Advisor learned for Player White, while Figures 24(b) and 24(c) show its corresponding Sitact Advisors formed by Create-Sitact-Advisor. Figure 24(b) contains the original context from Figure 24(a), paired with the first move in the original sequence. Figure 24(c) contains the context formed after executing White's first move in the sequence, and Black's subsequent move, paired with Player White's following move in the original sequence. In Figure 24(a) Player White learns to confine Player Black's piece in position

4, so that this Black piece cannot move. This is important in five men's morris since a player unable to move any piece loses the game.

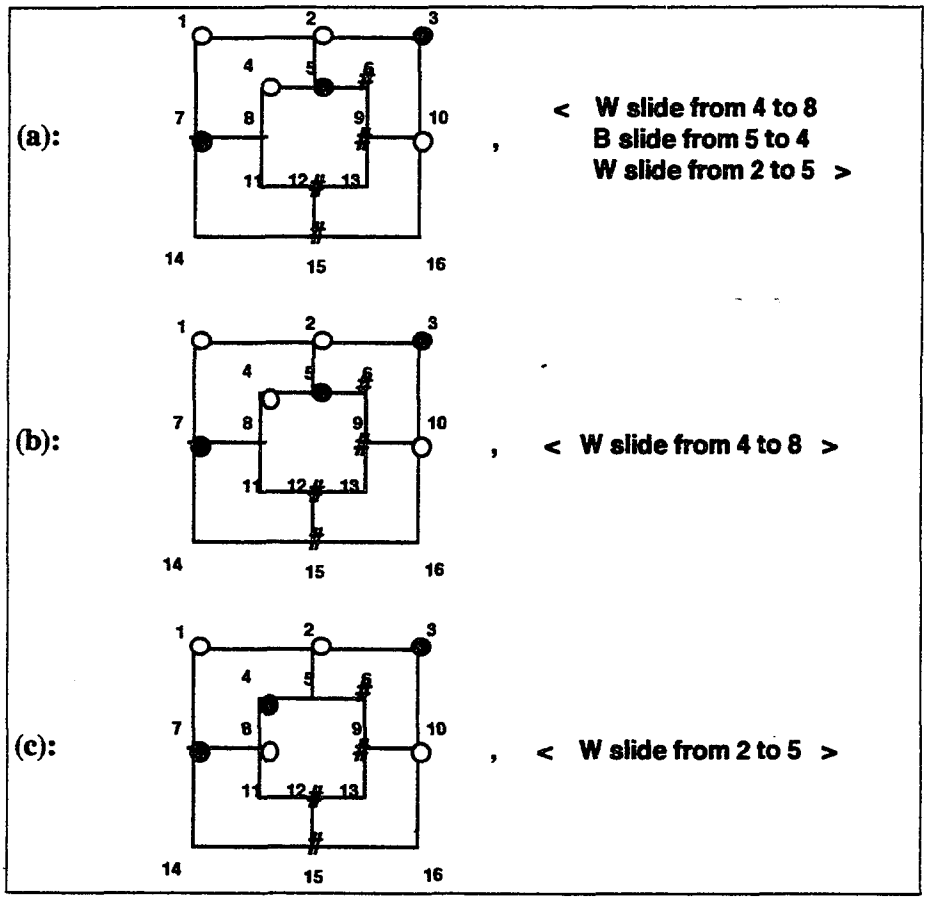


Figure24: (a) A five men's morris Sequence Advisor learned for Player White (b) and (c) the two corresponding Sitact Advisors

Case 4, in Figure 25, contains a five men's morris solo example, which requires more work than lose tic-tac-toe, since it must deal with sliding moves in addition to placing moves. Figure 25(a) contains a solo Sequence Advisor learned for Player White in five men's morris, where White attempts to take control of both sides of the board, by moving to the middle locations 8 and 9. The middle locations have access to three sliding moves, and four mills, so they are more powerful than the corner locations that have

access to two sliding moves and two mills. Figures 25(b)- 25(k) contain the ten Sitact Advisors corresponding to Figure 25(a). Figure 25(b) contains a Sitact Advisor, with the original context paired with the first move in the original sequence. To construct the others, Player White's move from 4 to 8 is executed, and all Black's legal moves within the pattern are generated and executed. In an abstracted Black sliding move, there are four possible types of moves (assuming that any # location can possibly instantiate to Black, White or empty).

Type 1) a slide from any # location to another # location.

Type 2) a slide from any Black location to an empty location

Type 3) a slide from any # location to an empty location

Type 4) a slide from any Black location to a # location

A Type 1 move does not affect the context, since all #'s remain as #, so all possible Type 1 moves lead to the formation of one Sitact Advisor. Figure 25(c) reflects an Advisor created after White slid from 4 to 8, and Black made a Type 1 move. Figures 25(d) - 25(f) are Advisors created after White slid from 4 to 8, and Black made a Type 2 move. Here there are three Type 2 moves: a slide from location 5 to location 4, a slide from location 15 to location 14, and a slide from location 15 to location 16, so three Advisors are formed. Figures 25(g) - (j) are created after White slid from 4 to 8 and Black took a Type 3 move. Here there are four possible Type 3 moves: a slide from location 2 to 3, a slide from location 7 to 14, a slide from location 10 to 16 and a slide from location 10 to 3. Figure 25(k) is created after White slid from 4 to 8 and Black took a Type 4 move. There is only one Type 4 move in this example: a slide from location 5 to location 2. In all, one Sequence Advisor turned into ten Sitact Advisors; had the original sequence been longer,

this process would be repeated looping down the sequence and creating even more Sitact Advisors.

For sequences without abstracted moves, the number of Sitact Advisors formed for each Sequence Advisor is the ceiling of (sequence length / 2). For example each Sequence Advisor with a length 3 sequence, leads to two Sitact Advisors, while each Sequence Advisor with a length five sequence leads to three Sitact Advisors. When using solo sequences, it is not possible to determine apriori how many Sitact Advisors will be formed because the number of possibilities for an abstracted move depends upon the formation of the pattern, and how much abstraction there is within the pattern. For example, while Figure 25(a) corresponds to ten Sitact Advisors, Figure 21(a) corresponds to two. This is true even though both Figure 21(a) and 25(a) suggest length three sequences. In practice, the number of Sitact Advisors formed was not as great as one might expect. Often, many of the legal instanstiations of an abstracted move were discarded because their execution prevented the next move in the sequence from being suggested as legal advice. In addition, most of the Sitact Advisors formed are not maintained to give advice. Just as the Sequence Advisors are weighted with PWL, the Sitact Advisors are weighted in further play using PWL also. Only those with high weights are maintained in the end.

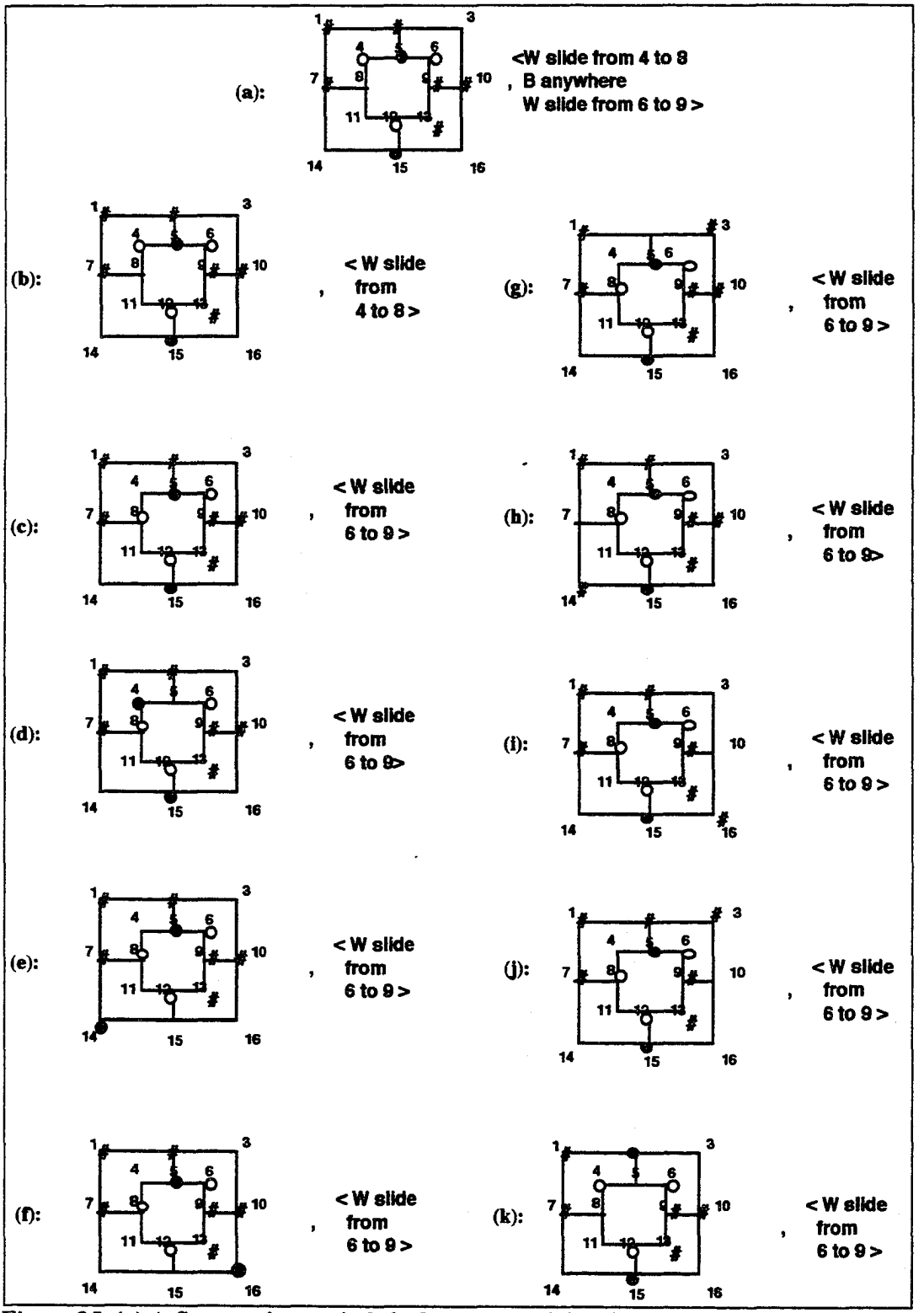


Figure 25: (a) A five men's morris Solo Sequence Advisor learned for Player White (b) a Sitact Advisor formed using (a)'s original context and pointing to the first move in (a)'s sequence (c) – (k) nine Sitact Advisors formed after Player White's first move in (a)'s sequence and all Player Black's possible subsequent moves are executed

4.3: Experimental Design

Since TP-Context's second experiment, which partitioned and segmented the Sequence Advisors, led to the best results, TP-Sitact mimicked its experimental design. SeqLearner had to be slightly revised to accommodate the change in representation. Since TP-Sitact no longer deals with sequences, there was no need for a tier-1 enforcement Advisor to implement a sequence set into motion. Instead, tier 2 and tier 3 were shifted upwards one tier. Tier 1 was partitioned into three segments; the top segment consisted of Advisors whose weights were from .9 to 1.0, the second segment of Advisors with weights from .8 to .89, and the bottom segment of Advisors with weights from .7 to .79. Tier 2 consisted of Advisors whose weights were from .55 to.69. If the two tiers did not choose a move, a move was chosen randomly.

An experiment was conducted on each of the same two games from Chapter 3: lose tic-tac-toe and five men's morris. Because SeqLearner is non-deterministic, breaking voting ties at random, each experiment reported here averaged data over twenty runs for lose tic-tac-toe and over ten runs for five men's morris. Each run consisted of a set of training tournaments, during which SeqLearner used TP-Sitact to learn about the game, followed by a set of testing tournaments that evaluated SeqLearner's game-playing ability against various contestants using the Sitact Advisors.

TP-Sitact experiments used the *double-cycle perfect vs. increasing* training environment. The training stage in each run consisted of eight watching tournaments. SeqLearner watched a random agent, then a 70%-reasonable agent (i.e., a novice), then a 10%-reasonable agent (i.e., an expert) and then a perfect agent compete, in that order, in a tournament against a perfect agent. SeqLearner then repeated these four watching

tournaments a second time. During this training stage, SeqLearner used TP-Sitact to learn <context, action> pairs and weight them accordingly. During training, the set of Sitact Advisors expands and contracts as their weights change. In each run, the training stage was followed by four 50-contest testing tournaments that pitted SeqLearner against the same three reasonable agents and a perfect agent. During testing, the set of Sitact Advisors in tiers 1 through 4 remained static since weights no longer change. All parameters remained as in TP-Context (Experiment II). For lose tic-tac-toe the watching tournaments consisted of 20 contests each, whereas for five men's morris they were of 40 contests each. In addition, for lose-tic-tac toe, sequences of length 3 were used to learn the Sitact Advisors, while for five men's morris sequences of length 3 and 5 were used.

4.4: Results

Table 11: Performance data for lose tic-tac-toe after 160 contests and for five mens morris after 320 contests (statistical improvements are highlighted in bold with standard deviations in parentheses below performance values)

	Perfect	Expert		Novice		Random	
	Reliability	**Reliability**	**Power**	**Reliability**	**Power**	**Reliability**	**Power**
Lose tic tac toe							
TP-Context: Exp II	82.00	83.80	12.10	85.10	46.70	87.00	57.40
	(13.24)	(9.73)	(4.22)	(5.08)	(7.08)	(4.17)	(7.02)
TP-Sitact	**91.00**	**90.30**	12.10	**89.70**	48.20	89.20	57.10
	(9.28)	(8.03)	(5.57)	(5.45)	(6.60)	(4.53)	(6.05)
Five men's morris							
TP-Context: Exp II	57.80	45.00	0.80	45.00	29.00	73.00	51.20
	(13.22)	(10.05)	(1.33)	(4.58)	(5.81)	(8.35)	(7.44)
TP-Sitact	**74.20**	**53.00**	0.40	46.00	28.00	69.80	50.20
	(7.56)	(7.11)	(0.80)	(10.08)	(9.03)	(7.61)	(10.56)

Table 11 compares the performance results of the TP-Context (Experiment II) and TP-Sitact experiments done with both lose tic-tac-toe and five men's morris. Areas where

TP-Sitact led to significant performance improvement are highlighted in bold. For lose tic-tac-toe TP-Sitact improved SeqLearner's reliability when playing against the perfect, expert and novice players, yet not when playing against the random player. This correlates with all the sequence methods studied thus far in this thesis; none are really applicable when playing against a random player which hops from move to move randomly without following sequences. For five men's morris, TP-Sitact improved SeqLearner's reliability against the perfect and expert players. The most striking result is the reliability of SeqLearner against the perfect five men's morris player after running TP-Sitact. Beginning with no knowledge other than observed sequences, SeqLearner is able to win and draw against a hand-coded perfect five men's player 74.20% of the time. TP-Sitact, however, does not improve TP-Context's power; the extra knowledge gained does not appear to be enough to make it to the winning move.

Figures 26(a) – 26(d) diagram how much better TP-Sitact is than TP-Context. For lose tic-tac-toe, Figure 26(a) compares the reliability of SeqLearner after running TP-Context (Experiment II) against its four opponents, with the reliability of SeqLearner after running TP-Sitact. The TP-Sitact gain in reliability is graphed in Figure 26(b). For five men's morris, these graphs are found in Figure 26(c) and 26(d). (Note that the graphs for the two games are drawn to a different scale.) Once again, it is clear that TP-Sitact is most useful for learning techniques against better opponents, as were TP-Context and TP-Rote.

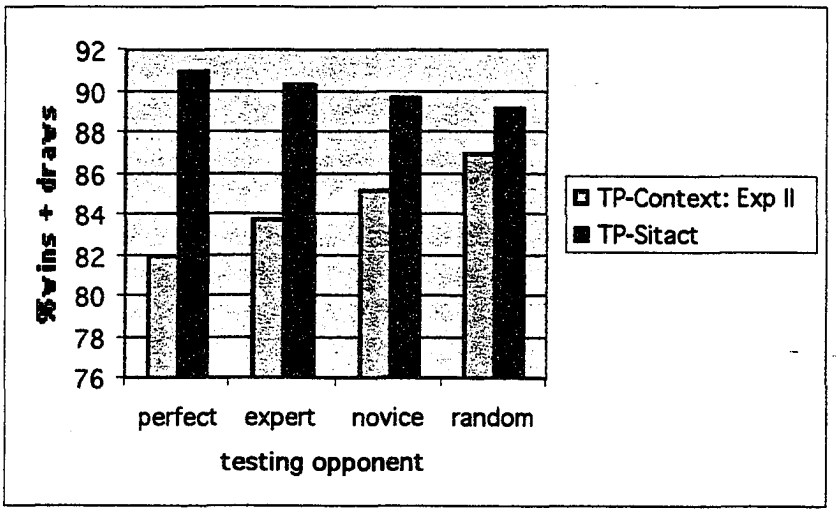


Figure 26(a): TP-Context's (Experiment II) reliability compared with TP-Sitact's reliability for lose tic-tac-toe.

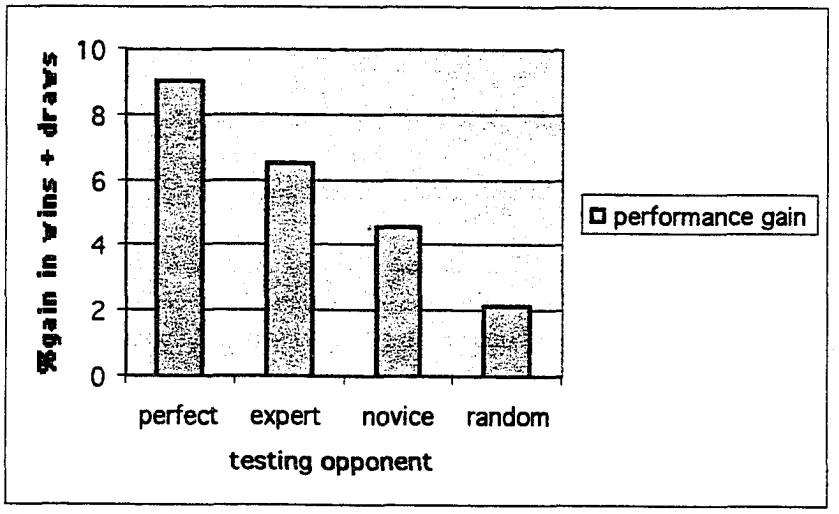


Figure 26(b): Performance Gain of TP-Sitact for lose-tic-tac-toe

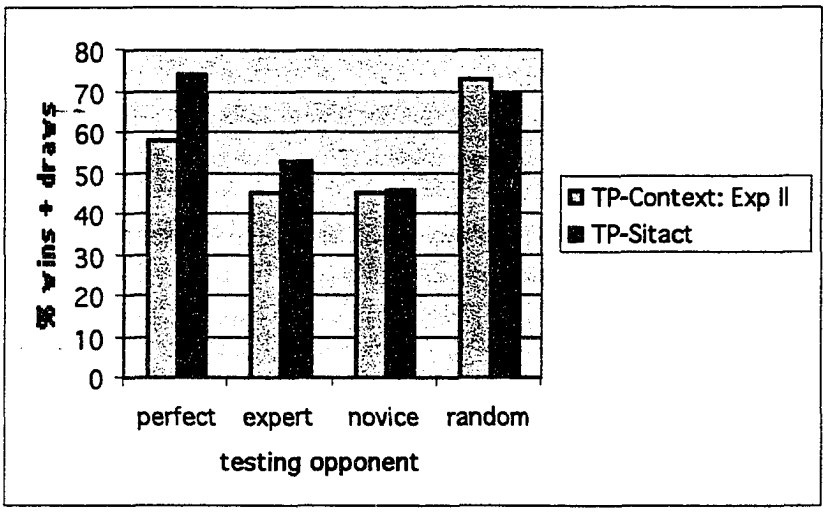


Figure 26(c): TP-Context's (Experiment II) reliability compared with TP-Sitact's reliability for five men's morris.

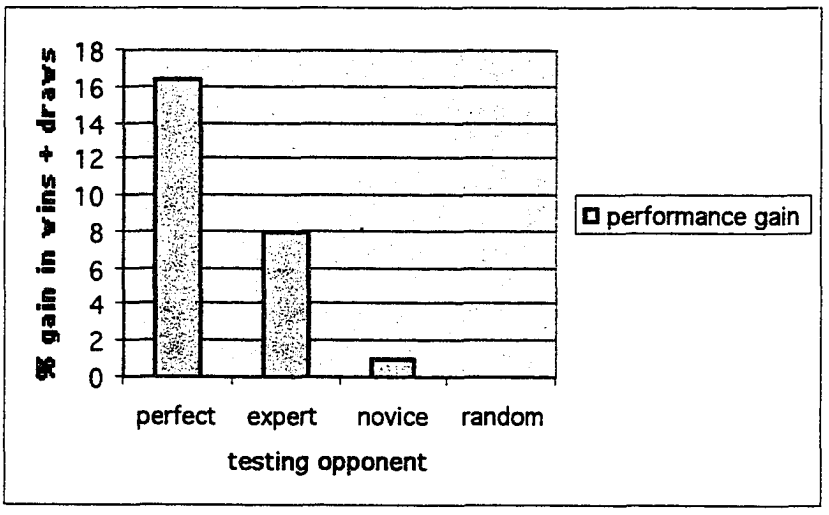


Figure 26(d): Performance Gain of TP-Sitact for five men's morris

It is important to examine how much more space and time is required by TP-Sitact versus TP-Context. Table 12 compares the number of context pairs spawned by each method and the percentage retained as Advisors after weighting the pairs in play.

Quite clearly the real growth occurs in the number of context pairs spawned. For lose tic-tac-toe, TP-Sitact spawned 2.16 times the number of context pairs. For five men's morris, TP-Sitact spawned 5.5 times the number of context pairs. The reason for the greater growth of context pairs with five men's morris is twofold: first, five men's morris has a greater number of move choices when TP-Sitact instantiates the abstracted moves in the solo sequences, and second because longer sequences were learned for five men's morris. (A longer sequence is more likely to convert into more Sitact Advisors than a shorter sequence.) However, the number of Advisors retained, which is of greater significance, does not grow that much for either game. For lose tic-tac toe, TP-Sitact learned 1.18 times the number of Advisors that TP-Context learned; for five men's morris, TP-Sitact learned 1.45 times the number of Advisors that TP-Context learned.

Table 12: Percentage of Context Pairs retained as Advisors

	#Context Pairs	std. dev.	# Advisors	std. dev.	%retained
Lose Tic Tac Toe					
TP-Context: Exp II	38.05	4.80	33.05	4.30	86.85%
TP-Sitact	82.25	10.13	39.30	3.47	47.78%
Five Men's Morris					
TP-Context: ExpII	3843.80	117.84	1615.60	77.77	42.03%
TP-Sitact	21157.90	1020.42	2344.40	81.36	11.08%

Table 13 displays the number of the Advisors learned in each level, for both TP-Context and TP-Sitact. For both games, TP-Sitact spawned the largest groups of Advisors in the tier 2's segment 1 and in tier 3. This offers a twofold advantage; first, the large group of excellent Advisors in segment 1 can make good decisions, second, the large

group of poor Advisors in tier 3 will rarely be reached in the decision-making process, thereby reducing execution time. The breakup into segments and tiers is similar for TP-Context and TP-Sitact; the one real difference is in tier 3. For lose tic-tac-toe, TP-Sitact's tier 3 is the second largest group instead of the smallest group as it was with TP-Context. For five men's morris, TP-Sitact's tier 3 is significantly larger than all three segments in tier 2, whereas TP-Context's tier 3 was the second largest group of Advisors.

Table 13: Breakdown of learned Advisors by weights

	Lose Tic-Tac-Toe		Five Men's Morris	
	TP-Context	**TP-Sitact**	**TP-Context**	**TP-Sitact**
Tier 2				
Segment 1 (weight .9 - 1.0)	17.80	25.45	575.50	670.80
Segment 2 (weight .8 - .89)	3.80	3.55	210.40	381.60
Segment 3 (weight .7 - .79)	6.65	2.60	277.90	471.40
Tier 3 (weight .55 -.69)	3.20	6.65	496.30	726.90

Table 14: Average Advisor usage and execution time during testing

	Advisor usage	**std dev.**	**execution time**	**std dev.**
Lose Tic Tac Toe				
TP-Context: Exp II	0.81	0.07	0.02	0.00
TP-Sitact	0.85	0.04	0.03	0.00
Five Men's Morris				
TP-Context: Exp II	0.89	0.01	2.55	0.20
TP-Sitact	0.91	0.01	5.51	0.25

Table 14 compares the average Advisor usage (i.e., the percentage of states during testing where SeqLearner had enough knowledge to make a non-random decision) and the average execution time in seconds that SeqLearner required per contest during

testing. (The machine and OS used for these experiments are described in Appendix B.) For both games there was a very small, yet significant increase in Advisor usage with TP-Sitact. The fact that this rise is so slight, even though TP-Sitact has many more Advisors, indicates that TP-Context's sequences are being followed, so that fewer Sequence Advisors can make almost as many decisions as more Sitact Advisors. The real drawback of the TP-Sitact extension is its increased execution time. Although lose tic-tac-toe's execution time is very quick for both methods there is still a significant 50% increase in execution time from .02 to .03. This result becomes clearer in the larger space of five men morris where TP-Sitact leads to a 116% increase in execution time. Because TP-Sitact must match its Advisors to each state, rather than following a sequence for a few moves as TP-Context can, it requires more execution time. Hence the tradeoff: faster execution with TP-Context versus better performance with TP-Sitact. If the performance improvement is significant, the tradeoff in time is worth it.

4.5: Discussion

TP-Sitact has proved to be a successful extension to TP-Context. It significantly improves performance, albeit requiring more execution time. There are two reasons why TP-Sitact leads to stronger performance. The first is that breaking up the Advisors into separate entities allows them to be applicable more frequently as shown by the greater Advisor usage in the TP-Sitact experiment. When dealing with sequences, a move down in the sequence can only be recommended if its context arose via the sequence. If this same context happened to arise via a different path in the game tree a move cannot be suggested. Since the Sitact Advisors pair each action with a context, they can apply even if a context arose via a different path than the sequence it was learned from. However,

because Advisor usage only rises slightly from TP-Context to TP-Sitact, this does not seem to the main strongpoint of TP-Sitact. In fact, so small a rise is a good indicator that there do exist sequences that are important and that the contexts within these sequences will usually arise in these same paths. This correlates to the result in Chapter 2, Section 2.4, where the Imitate extension (which converted TP-Rote from a situation-sequence representation to a situation-action representation) was not used much more than TP-Rote.

The second reason that TP-Sitact leads to better play is more significant. Consider the TP-Context Sequence Advisor learned for Player X in lose tic-tac-toe, displayed in Figure 22(a). Suppose using this Advisor that the first move it suggests, <X to 1>, is correct, but that the second move it suggests, <X to 7>, is not. Because TP-Context associates a sequence with only one weight, this Advisor's weight will oscillate; it will rise when the first move is suggested, and fall when the second move is suggested. TP-Sitact, on the other hand, separates the actions and can therefore weight them separately, allowing the weighting process to work better. In such cases as the above example, it can keep parts of the sequence that are correct and discard the others. TP-Sitact's instantiation of the solo sequences also helps weed out incorrect behavior. Suppose, using Figure 25, that some of Player Black's moves cause Player White's next move (slide from 6 to 9) to be correct and some do not. The Sitact Advisors have the ability to learn this and retain only the correct advice.

Still, it is important to remember that the Sitact Advisors cannot be created without first creating the Sequence Advisors. Organizing observed sequences is TP-Context's underlying key to honing in on context, and only then can the representation be

shifted. In addition, collecting solo sequences allows TP-Context to learn the context for a one-player plan, and then TP-Sitact can instantiate it to pinpoint when the plan pertains and when it does not, depending upon the opponent's play. If TP-Context were to look at all the opponent's moves to begin with, however, the context for a one-player plan would never be reached. The abstraction is required before it can be eliminated.

TP-Sitact scaled well from lose tic-tac-toe to five men's morris. The main growth was in the number of context pairs spawned rather than in the number of Advisors retained. This should allow it to scale further, since ultimately the amount of space and time required to play a game depends on the number of Advisors retained. If the number of context pairs grows too much, TP-Sitact can be revised to occasionally discard the very low weight context pairs along the way.

Chapter 5: Related Work and Discussion

5.1: Related Work

There is a large body of research related to the three methods studied in this thesis. TP-Rote's idea of composing a sequence of operators together is not new; it was first implemented as MACROPs for STRIPS, and later extended by Korf's macro-operators (Fikes, Hart et al. 1972; Korf 1985). Unlike TP-Rote, however, previous research focused on single-agent domains. MACROPs were intended to increase the efficiency of the General Problem Solver, which used means-ends-analysis to solve problems. The system was limited to domains where the goal state was a conjunction of well-defined subgoals, and which possessed a set of operators with well-defined preconditions and postconditions. Game playing is not such a domain; a game may have no set of defined subgoals, and actions can have subtle, far-reaching effects that are not clearly delineated.

Korf extended the STRIPS approach; he built a method to learn a complete set of macro-operators for certain problems, one that eliminated the need for search entirely. His approach, however, was restricted to problems where the sets of subgoals were *serially decomposable*; that is, there was an ordering of the subgoals such that the solution to each subgoal depended only on that specific subgoal and the ones preceding it in the ordering. Game playing fails to fit in this category. More recently, Micro-Hillary used a heuristic function to selectively learn macros that lead from a local minimum to a better state (Finkelstein and Markovitch 1998). This was also implemented for single agent domains. For more than one agent, a macro's utility depends upon more than leading to a better state; it depends upon the likelihood that the other agent will follow the sequence of actions too.

SOAR uses a knowledge compilation mechanism called *chunking* to learn (Laird, Rosenbloom et al. 1986). Chunks compile the results of subgoal search into one production rule that can produce the same result without search. SOAR has been applied to tic-tac-toe. Unlike TP-Rote and TP-Context, however, the learned chunks do not result in compiled sequences of moves that occur often. The chunks learned by SOAR are fairly general rules that test the board for certain conditions and select a single, appropriate move. SOAR also relies on a complete domain theory, whereas none of the methods in this thesis do.

Like case-based planners, TP-Context indexes a sequence under a context to retrieve for future use. Case-based planners, however, also typically use a set of hand-chosen features as the context for a sequence, while TP-Context's contexts are learned. In addition, unlike TP-Context, case-based planners require large amounts of domain knowledge (Hammond 1989; Hammond, Converse et al. 1993). Although TP-Context extrapolates a context from limited experience inductively, searching more broadly would be costly. TP-Context takes a computationally reasonable approach; once it narrows the focus, a deductive method can be used to further refine the knowledge it learns. For example, a narrow search from an induced context could be attempted to prove correctness.

Like Finkelstein and Markovitch's Sylph, TP-Context learns and uses abstract patterns during experience (Finkelstein and Markovitch 1998). Unlike Sylph's chess patterns, which are associated with individual moves and used for move evaluation, TP-Context's patterns are associated with sequences of moves and used for move selection. Sylph's representation of chessboards incorporates both static and dynamic relations.

Future work could extend TP-Context's pattern representation to enable it to reason about why a specific pattern was associated with a particular move sequence. Such reasoning capabilities would allow TP-Context to ensure a sequence's applicability while executing it in a novel situation.

Sequential Instance Based Learning (SIBL) and TP-Context are both methods that learn from sequences of decisions made in prior experience (Epstein and Shih 1998). Because both methods exploit the knowledge contained in these sequences, they require little other domain knowledge. They differ, however, both in their learning approach and in their usage of learned knowledge. SIBL is an extension of Instance Based Learning that learns to appropriately select actions in new bridge contests based on action patterns (i.e., sequences of consecutive states) played in previous contests (Aha, Kibler et al. 1991; Epstein and Shih 1998). Rather than learn the context for a sequence as TP-Context does, SIBL treats the sequence that led to an action as the context for the action itself, and trains a database of <sequence, action> pairs based on similarity metrics that measure a new sequence's similarity to known sequences. TP-Context organizes its experience based upon sequences seen to learn <context, sequence> pairs rather than <sequence, action> pairs.

Kojima's evolutionary algorithm that learns patterns for Go, discussed in Section 1.3.4, can learn sequences of moves (Kojima 1998). Patterns are represented as production rules with the *If* part consisting of conditions on the board, and the *Then* part consisting of an action. The possible conditions that can be randomly chosen when building a rule include stone color, edges, and previous moves. Allowing previous moves to be conditions in the pattern enables Kojima's system to learn sequences as part of the

context for the next single action to take. In this way Kojima's system is similar to SIBL, and different from TP-Context, which learns sequences to be suggested as plans to follow.

Other attempts to build game-playing programs that rely on pattern-knowledge include the PARADISE and Morph systems described in Chapter 1. Like TP-Context, PARADISE exploits patterns to reduce search, and uses patterns to form plans. However, unlike the methods studied in this thesis, PARADISE was provided with a manually constructed pattern-database, and did not use learning to automatically acquire these patterns. Morph, on the other hand, like TP-Context, automatically acquires patterns, and learns weights associated with these patterns. It differs from TP-Context both in its method of acquiring patterns and in their usage. Morph, unlike TP-Context, does not use sequence knowledge to learn context. Furthermore, patterns learned are neither associated with moves, nor with sequences. Instead, Morph suggests a move based upon the combination of pattern weights of the patterns in the state that a move forms. This is more similar to Patsy than to TP-Context, which selects a move based upon the sequences that the patterns found in a given state advise.

EBG has also been applied in the game-playing domain to learn patterns and plans. Among such efforts are Minton's application to Go-Moku and, more recently, Kojima's application to the game of Go. Minton learned plans in Go-Moku for forcing states by backing up from a state where the goal of five-in-a-row was reached. Similarly, Kojima learned plans in Go for the capture of stones by backing up from states where stones were captured. In both these systems the programs do not find the forcing states; they are given them. Both systems are limited to learning patterns and plans for tactical combinations

where each of the opponent's moves are forced. The methods in this thesis have no such restriction and can discover other interesting patterns.

5.2: Discussion

Although all the methods outlined in this thesis learn temporal patterns, they differ in their purposes, advantages and limitations. TP-Rote is a rote-learning method that hones in on frequently used segments of a search space and compactly memorizes them for later use. Experiments proved TP-Rote's ability to save time while conserving space. TP-Rote learned opening and endgame sequences, as well as others. Because TP-Rote considers only the limited number of states seen during experience, and neither searches nor reasons, its algorithm can be applied in any size domain. Its success in decreasing execution time, however, depends upon how many stereotyped sequences a domain possesses, and how often these sequences are exploited. TP-Rote's main limitation is its inability to generalize states, which limits sequence usage to states that have already been seen.

TP-Context exploits the knowledge inherent in experienced sequences to aid context discovery. Each sequence seen in a contest is associated with the set of states upon which it was executed, thereby narrowing the discovery problem to a small set of states from which the context can be extracted. Unlike other pattern-learning programs that learn patterns associated with moves, TP-Context learns patterns associated with sequences. Examination of the sequences learned show that TP-Context learned contexts for opening and endgame sequences along with other important midgame sequences. Because TP-Context generalizes states, it expands the applicability of the knowledge it learns to novel states, overcoming TP-Rote's limitation. Empirical results show that TP-

Context can learn a lot about playing two games using this approach, proving that sequences are an important knowledge source for learning programs. The fact that TP-Context induces contexts from the limited number of experienced sequences should enable it to scale to larger spaces. However, in a larger domain, quite a great number of training examples will be needed for TP-Context's induction to be correct. It therefore might be necessary to add some domain-specific knowledge to guide the sequence-gathering of TP-Context.

TP-Sitact extends the TP-Context method, by shifting its situation-sequence representation to a situation-action representation. Results show that while TP-Sitact requires more time and space, it significantly improves TP-Context's performance. TP-Sitact should scale to larger spaces because the growth in the number of Advisors it retains over the number TP-Context retains is not that large. Although both TP-Context and TP-Sitact improve Seq-Learner's reliability, neither method improves Seq-Learner's power. In order to win a game, knowledge other than sequence knowledge is needed. This indicates that a sequence-oriented learning program should be added as a component in a multiple-method learning program rather than be used alone.

Table 15 is one way to compare the various methods researched in this thesis. Machine learning's challenge is to learn to respond intelligently when faced with a situation. TP-Rote and Imitate require each situation to be represented as the complete state of the world at that time. TP-Context and TP-Sitact, on the other hand, represent each situation in a generalized form. Whereas TP-Rote and TP-Context learn to respond to a situation with a move sequence, Imitate and TP-Sitact learn to respond with an individual move. It is important to note that Imitate and TP-Sitact are extensions of TP-

Rote and TP-Context, respectively, rather than independent algorithms. Imitate and TP-Sitact use the knowledge learned by TP-Rote and TP-Context, yet shift the representation from situation-sequence to situation-action. While Imitate requires a lot more space than TP-Rote, it leads to greater execution time speedup. Similarly, while TP-Sitact requires more time and space than TP-Context, it leads to improved performance. In all, TP-Rote's goal of decreasing execution time is improved with Imitate, while TP-Context's goal of learning to play a game is improved with TP-Sitact. One can conclude that converting from a situation-sequence representation (after sequences are learned) to a situation-action representation can bring a goal closer, but likely at the expense of more computer resources.

Table 15: Situation-Response View of the World

	SITUATION	RESPONSE
1: TP-ROTE	state	move sequence
2: IMITATE	state	move
3: TP-CONTEXT	generalization	move sequence
4: SITACT	generalization	move

Recall that TP-Rote was built on top of a pre-existing game-playing program, Hoyle, to decrease Hoyle's execution time with learned expertise. TP-Context and TP-Sitact, on the other hand, were implemented independently from Hoyle, to see what and

how much these methods can learn about a game from experienced sequences. Much of what TP-Context and TP-Sitact learn is already encapsulated in Hoyle's hand-coded Advisors, and adding these two methods to Hoyle would mask what TP-Context and TP-Sitact can learn without this prior knowledge. However, once TP-Context and TP-Sitact were fully implemented, some experiments were run with lose-tic-tac-toe and five men's morris, to see if adding the knowledge they learn to Hoyle could in some way improve Hoyle's performance.

Hoyle's architecture has a top tier of perfectly correct Advisors that are consulted sequentially in a pre-specified order. If a move is suggested by a tier-1 Advisor, it is taken; if not control flows to the second tier of Advisors. The tier-2 Advisors offer heuristic, possibly conflicting advice. A vote is then taken to determine which move has the most support and should be selected. To add TP-Sitact to Hoyle, a new Advisor, Seq-Advisor, was created and added to Hoyle's second tier. Two experiments (each averaged over ten runs) were run for each game; one using standard Hoyle and one using Hoyle with Seq-Advisor. All the experiments used the *double-cycle perfect vs. increasing* training environment, where the training stage in each run consisted of eight watching tournaments. Hoyle watched a random agent, then a 70%-reasonable agent (i.e., a novice), then a 10%-reasonable agent (i.e., an expert) and then a perfect agent compete, in that order, in a tournament against a perfect agent. Hoyle then repeated these four watching tournaments a second time. During this training stage, Hoyle's weights and useful knowledge were learned. Training was followed by four 50-contest testing tournaments that pitted Hoyle against the same three reasonable agents and a perfect agent. In the experiments where Seq-Advisor was added to Hoyle's second tier, the Sitact

Advisors were learned during training along with all the other knowledge that Hoyle learns. During testing, Seq-Advisor accessed all Sitact Advisors of weight 1.0 in an attempt to add to Hoyle only the most reliable knowledge that TP-Sitact had learned. Seq-Advisor combined the advice and weights of these Sitact Advisors in a vote to choose one move to suggest. This move was then voted upon with the rest of Hoyle's tier-2 Advisors.

Table 16: Performance data for lose tic-tac-toe after 160 contests and for five mens morris after 320 contests (statistical improvements are highlighted in bold with standard deviations in parentheses below performance values)

	Perfect	**Expert**		**Novice**		**Random**	
	Reliability	**Reliability**	**Power**	**Reliability**	**Power**	**Reliability**	**Power**
Lose tic tac toe							
standard Hoyle	99.00	97.00	18.20	97.60	64.00	98.80	73.20
	(2.05)	(5.16)	(4.85)	(1.96)	(6.87)	(0.98)	(6.94)
Hoyle + Seq-Advisor	99.20	98.80	16.40	98.00	62.20	98.80	74.20
	(2.40)	(1.60)	(4.54)	(1.55)	(6.23)	(1.83)	(3.63)
Five men's morris							
standard Hoyle	91.40	84.80	16.00	92.40	71.40	96.60	87.80
	(3.89)	(4.34)	(2.86)	(4.20)	(4.90)	(3.13)	(4.47)
Hoyle + Seq-Advisor	**96.20**	83.40	15.20	90.20	72.80	96.60	88.00
	(3.32)	(3.41)	(4.90)	(3.71)	(7.44)	(2.32)	(5.33)

Table 17: Average execution time during testing (in seconds)

	execution time	**std dev.**
Lose Tic Tac Toe		
standard Hoyle	0.25	0.07
Hoyle + Seq-Advisor	0.25	0.02
Five Mens Morris		
standard Hoyle	13.68	6.30
Hoyle + Seq-Advisor	18.09	9.62

Results of the experiments are shown in Table 16. For five men's morris, adding Seq-Advisor significantly improved Hoyle's reliability against the perfect player. This reveals TP-Sitact's ability to capture important game-dependent information that is neither encapsulated in Hoyle's game-independent Advisors, nor among Hoyle's other learned useful knowledge. Furthermore, in two out the ten runs, Hoyle with Seq-Advisor achieved 100% reliability against the perfect player, which Hoyle on its own was never able to achieve. For all other opponents in both lose-tic-tac-toe and five men's morris, Seq-Advisor made no significant difference in Hoyle's performance. This means that the small amount of knowledge that Hoyle is missing is not captured by TP-Sitact's highest-weighted sequence knowledge. Table 17 displays the change in execution time caused by adding Seq-Advisor to Hoyle. For lose-tic-tac-toe there is no change. For five men's morris, where there are many more Sitact Advisors of weight 1.0 that must be accessed, execution time during testing increases from 13.68 seconds to 18.09 seconds.

Initial attempts to add Seq-Advisor to tier 1 were not successful because the Sitact Advisors are learned inductively and are not perfect like the other tier 1 Advisors. Adding Seq-Advisor as a middle tier between Hoyle's top and bottom tiers also did not lead to improved five men's morris performance, although adding Seq-Advisor to Hoyle's tier 2 did lead to improved performance. This can be understood by examining the weight that Seq-Advisor achieved as a tier-2 Hoyle Advisor. For five men's morris, in the placing stage, Seq-Advisor received the fifth highest weight out of 14 Advisors, while in the sliding stage it received the fourth highest weight. Anthropomorph, Material, and Vulnerable received higher weights than Seq-Advisor in both the placing and sliding stages, while Freedom achieved a higher weight than Seq-Advisor only in the placing

stage. Anthropomorph is an Advisor that imitates a perfect player's moves when a previously seen state re-arises. Material is coded to suggest a move that captures the other player's pieces, while Vulnerable is coded to suggest a move that prevents Hoyle's pieces from being captured. Freedom suggests moves that maximize the number of its subsequent immediate moves or minimize those of the non-mover. Since these concepts are crucial in five men's morris, they receive higher weights than Seq-Advisor. Giving Seq-Advisor's inductively acquired sequence knowledge priority over tier-2 Advisors that achieve higher weights than Seq-Advisor did not lead to improved performance.

Neither TP-Rote nor TP-Context is restricted to two-agent-domains; both methods could learn sequences that involve one agent or more than two agents. TP-Rote's success, as previously mentioned, depends on how many stereotyped sequences a domain possesses. TP-Context's approach will work best in domains that possess strategies that have contextual dependence. As implemented here, TP-Context is only for domains where geometric patterns motivate subsequent actions, so that the context for a sequence of actions is a pattern. A different context extraction method could apply TP-Context to a domain where actions are dependent upon some context not describable as a geometric pattern.

Meanwhile the work outlined in this thesis has demonstrated that temporal patterns are an important resource for programs learning expertise in a particular domain. While many researchers have used patterns to aid decision making, and others have explored ways to learn the significant patterns in a domain, this work focuses on sequences of actions that create patterns. Future work will experiment with other domains, other context

extraction methods, and with using domain-specific knowledge to guide the sequence gathering of TP-Context, rather than gathering sequences of various lengths.

Appendix A : The Games and Experimental Environment

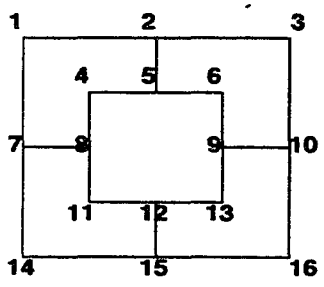
Lose-Tic-Tac-toe

Board: 3x3 grid

Rules: Player-1 has five X's and Player-2 five O's. Initially the board is empty. A turn consists of placing one of your markers in an empty square. The first one to place three owned markers in a row, vertically, horizontally or diagonally, loses. There are eight such losing lines. Play ends in a draw when there are no more empty squares.

Five-Men's-Morris

Board:



Rules: Player-1 has five black markers and Player-2 has five white ones. Initially the board is empty and a turn consists of placing one of your markers on the intersection of two or more lines. There are 16 such locations. Once all five of your markers are on the board, a turn consists of sliding one of your markers along a line to the next empty location. You may not jump over another marker or lift your marker from the board during a slide. Three of the same markers in a straight line along any side of the small square or the large square are called a mill. Each time a player gets a mill, she removes one of the other player's markers that is not in a mill. If the other player's markers are all in mills, however, she removes one from a mill. The first one reduced to two markers or unable to move loses.

Experimental Environment:

- PowerMac G3/ 400 Megahertz
- Mac OS 8.5
- Mac OS ROM 1.2

Appendix B: Hoyle's Advisors for Game Playing.

Name	Tier	Description
Wiser	1	Makes the correct move if the current state is remembered as a certain win.
Sadder	1	Resigns if the current state is remembered as a certain loss.
Victory	1	Makes the winning move from the current state if there is one.
Don't Lose	1	Eliminates any move that results in an immediate loss.
Panic	1	Blocks a winning move the non-mover would have if it were his turn now.
Shortsight	1	Advises for or against moves based on a two-ply lookahead.
Enough Rope	1	Avoids blocking a losing move the non-mover would have if it were his turn now.
Anthropomorph	2	Moves as a winning or drawing non-Hoyle expert did.
Candide	2	Formulates and advances naive offensive plans.
Challenge	2	Moves to maximize its number of winning lines or minimize the non-mover's.
Coverage	2	Maximizes the mover's markers' influence on predrawn game board lines or minimizes the non-mover's.
Cyber	2	Moves as a winning or drawing Hoyle did.
Greedy	2	Moves to advance more than one winning line.
Leery	2	Avoids moves to a state from which a loss occurred, but where limited search proved no certain failure.
Material	2	Moves to increase the number of its pieces or decrease those of the non-mover.
Freedom	2	Moves to maximize the number of its subsequent immediate moves or minimize those of the non-mover.
Not Again	2	Avoids moving as a losing Hoyle did.
Pitchfork	2	Advances offensive forks or destroys defensive ones.
Vulnerable	2	Reduces the non-mover's capture moves on two-ply lookahead.
Worried	2	Observes and destroys naive offensive plans of the non-mover.

Bibliography

Aha, D., D. Kibler, et al. (1991). "Instance-Based Learning Algorithms." *Machine Learning* **6**: 37-66.

Allen, J., E. Hamilton, et al. (1997). New Advances in Adaptive Pattern-Oriented Chess. *Advances in Computer Chess.* H. J. Herik and J. W. Uiterwijk. Maastricht, The Netherlands. **8:** 213-233.

Amarel, S. (1968). On the representation of problems of reasoning about actions. *Machine Intelligence.* D. Michie. New York, American Elsevier.

Anzai and H. Simon (1979). "The Theory of Learning by Doing." *Psychological Review* **36**(2).

Ash, T. (1989). "Dynamic Node Creation in Backpropagation Networks." *Connection Science* **1**: 365-375.

Berliner, H. (1979). *On The Construction Of Evaluation Functions For Large Domains.* IJCAI, Tokyo, Japan.

Berliner, H. (1980). "Backgammon Computer Program Beats World Champion." *Artificial Intelligence* **14**: 205-220.

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

Burmeister, J. (2000). Studies in Human and Computer Go: Assessing the Game of Go as a Research Domain for Cognitive Science. *Department of Electrical Engineering and Computer Science and Department of Psychology.* Australia, The University of Queensland.

Buro, M. (1995). "Probcut: An effective selective extension of the alpha-beta algorithm." *ICCA Journal* **20**(3): 71-76.

Buro, M. (1997). "The Othello match of the year: Takeshi Murakami vs. Logistello." *ICCA Journal* **20**(3): 189-193.

Buro, M. (1999). "Toward Opening Book Learning." *ICCA Journal* **22**(2): 98-102.

Buro, M. (2002). "Improving Heuristic Mini-Max Search by Supervised Learning." *Artificial Intelligence* **124**((1-2)): 85-99.

Campbell, M. (1999). "Knowledge Discovery in Deep Blue." *Communications of the ACM* **42(11)**: 65-67.

Campbell, M., A. J. Hoane, et al. (2002). "Deep Blue." *Artificial Intelligence* **134**: 57-83.

Chase, W. G. and H. A. Simon (1973). The Mind's Eye in Chess. *Visual Information Processing*. W. G. Chase. New York, Academic Press: 215-281.

Chase, W. G. and H. A. Simon (1973). "Perception in Chess." *Cognitive Psychology* **4**(1): 55-81.

Cohen, D. I. A. (1972). The Solution of a Simple Game. *Mathematics Magazine*. **45**: 213-216.

De Groot, A. D. (1966). Perception and memory versus thought: some old ideas and recent findings. *Problem Solving*. B. Kleinmuntz. New York, Wiley.

Epstein, S. and J. Shih (1998). *Sequential Instance-Based Learning*. Proceedings of the AI-98, Vancouver.

Epstein, S. L. (1994a). "For the Right Reasons: The FORR Architecture for Learning in a Skill Domain." *Cognitive Science* **18**(3): 479-511.

Epstein, S. L. (1994b). *Identifying the Right Reasons: Learning to Filter Decision Makers*. Proceedings of the AAAI 1994 Fall Symposium on Relevance, Palo Alto.

Epstein, s. L. (1995). "Learning in the Right Places." *The Journal of the Learning Sciences* **4**(3): 281-319.

Epstein, S. L., J. Gelfand, et al. (1996). Spatially-Oriented Agents Improve A Multi-Agent Decision-Making Program. *Spatial and Temporal Reasoning Workshop, Thirteenth National Conference on Artificial Intelligence*. Portland, Oregon: 5-13.

Ericsson, K. A. and J. Smith (1991). Prospects and Limits of the Empirical Study of Expertise: An Introduction. *Toward a General Theory of Expertise - Prospects and Limits*. K. A. Ericsson and J. Smith. Cambridge, Cambridge University Press: 1-38.

Fikes, R. E., P. E. Hart, et al. (1972). "Learning and Executing Generalized Robot Plans." *Artificial Intelligence* **3**: 251-288.

Finkelstein, L. and S. Markovitch (1998). "Learning To Play Chess Selectively By Acquiring Move Patterns." *ICCA Journal* **21**(2): 100-119.

Finkelstein, L. and S. Markovitch (1998). "A Selective Macro-learning Algorithm and its Application to the N x N Sliding-Tile Puzzle." *Journal of Artificial Intelligence Research* **8**: 223-263.

Fuernkranz, J. (2001). Machine Learning in Games: A Survey. Huntington, NY, Nova Science Publishers.

Ginsberg, M. L. (1998). Computers, Games and the Real World. *Scientific American Presents Special Issue on Exploring Intelligence.*

Greenblatt R. D., Eastlake III D. E., et al. (1967). *The Greenblatt Chess Program*, American Federation of Information Processing Societies.

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

Hammond, K. J., T. Converse, et al. (1993). "Opportunism and Learning." *Machine Learning* **10**: 279-309.

Holding, D. (1985). *The Psychology of Chess Skill.* Hillsdale, NJ, Lawrence Erlbaum.

Iiada, H., M. Sakuta, et al. (2002). "Computer Shogi." *Artificial Intelligence* **134**: 121-144.

Koenig, S., D. Furcy, et al. (2002). *Heuristic Search-Based Replanning*. Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling.

Koenig, S. and M. Likhachev (2002). *Improved Fast Replanning for Robot Navigation in Unknown Terrain.* Proceedings of the International Conference on Robotics and Automation.

Kojima, T. (1998). Automatic Acquisition of Go Knowledge from Game Records: Deductive and Evolutionary Approaches. *Graduate Division of International and Interdisciplinary Studies.* Tokyo, Japan, The University of Tokyo: 91.

Kojima, T. and A. Yoshikawa (1999). *Knowledge Acquisition From Game Records.* ICML Workshop on Machine Learning in Game Playing.

Korf, R. (1985). "Macro-Operators: A Weak Method for Learning." *Artificial Intelligence* **26**: 35-77.

Laird, J., P. S. Rosenbloom, et al. (1986). "Chunking in SOAR: The Anatomy of a General Learning Mechanism." *Machine Learning* **1**(11-46).

Lee, K. F. and S. Mahajan (1990). "The Development of a World Class Othello Program." *Artificial Intelligence* **43**(1): 21-36.

Levinson, R. (1996). "General Game-Playing and Reinforcement Learning." *Computational Intelligence* **12**(1): 155-176.

Levinson, R. and R. Snyder (1991). *Adaptive Pattern-Oriented Chess*. Proceedings of the Eighth International Machine Learning Workshop, Morgan Kaufman.

Levinson, R. and R. Weber (2000a). *Chess Neighborhoods, Function Combination, and Reinforcement Learning*. CG2000: Second International Conference on Computers and Games.

Markovitch, S. and P. D. Scott (1993). "Information Filtering: Selection Mechanisms in Learning Systems." *Machine Learning* **10**: 113-151.

Matsubara, H., H. Iiada, et al. (1996). "Natural Developments in Game Research." *ICCA Journal* **19(2)**: 103-112.

Michie, D. (1974). *On Machine Intelligence*. Edinburgh, Edinburgh University Press.

Minsky, M. (1963). Steps toward artificial intelligence. *Computers and Thought*. E. Feigenbaum and J. Feldman. New York, McGraw-Hill.

Minton, S. (1984). *Constraint-based generalization - Learning game-playing plans from single examples*. Proceedings of the Fourth National Conference on Artificial Intelligence, William Kaufmann.

Mitchell, T. M. (1997). *Machine Learning*, McGraw-Hill Companies, Inc.

Mitchell, T. M., R. M. Keller, et al. (1986). "Explanation-Based Generalization: A unifying view." *Machine Learning* **1**(1): 47-80.

Muller, M. (2000). *Not Like Other Games - Why Tree Search in Go is Different*. Proceedings of the Fifth Joint Conference on Information Sciences (JCIS 2000).

Muller, M. (2002). "Computer Go." *Artificial Intelligence* **134**: 145-179.

Reitman, W. and B. Wilcox (1979). *The Structure and Performance of the Interim.2 Go Program*. IJCAI-79, Tokyo, Japan.

Russell, S. and P. Norvig (1995). *Artificial Intelligence: A Modern Approach*.

Samuel, A. L. (1959). "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development* **3**: 210-259.

Samuel, A. L. (1967). "Some Studies in Machine Learning Using the Game of Checkers II - Recent Progress." *IBM Journal of Research and Development* **11**: 601-617.

Schaeffer, J. (1997). *One Jump Ahead*, Springer-Verlag.

Schaeffer, J. (2000). The Games Computers (and People) Play. *Advances in Computers.* M. V. Zelkowitz, Academic Press. **50.**

Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine.* **41:** 256-275.

Slate, D. (1987). "A Chess Program that uses its Transposition Table to Learn from Experience." *Journal of the International Computer Chess Association* **10(2)**: 59-71.

Tesauro, G. (1995). "Temporal difference learning and TD-Gammon." *Communications of the ACM* **38**(3): 58-68.

Treisman, A. (1986). Features and Objects in Visual Processing. *Scientific American:* 114-125.

Utgoff, P. and D. Precup (1997). Constructive Function Approximation, University of Massachusetts.

Wilkins, D. (1980). "Using Patterns and Plans in Chess." *Artificial Intelligence* **14**: 165-203.

Wynne-Jones, M. (1992). "Node splitting: a constructive algorithm for feed-forward neural networks." *Advances in Neural Information Processing Systems*: 1072-1079.