10-18-2007

The Structure of Games

David Michael Kaiser

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

THE STRUCTURE OF GAMES

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

David M. Kaiser

2007

To: Interim Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by David M. Kaiser, and entitled The Structure of Games, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

_____
Malek Adjouadi

_____
Geoffrey Smith

_____
Ana Pasztor

_____
Tao Li, Major Professor

Date of Defense: October 18, 2007

The dissertation of David M. Kaiser is approved.

_____
Interim Dean Amir Mirmiran
College of Engineering and Computing

_____
Dean George Walker
University Graduate School

Florida International University, 2007

DEDICATION

This dissertation is dedicated to my angelic wife Adrienne. Her steadfast love, endless support and unwavering encouragement made the completion of this work possible. This dissertation is also dedicated to my children Alexander and Katherine. Get yours done early rather than later.

ACKNOWLEDGMENTS

First and foremost I thank my wife Adrienne. Although she has had cause to regret suggesting a Ph.D. in the first place, she has never wavered in her support. I also thank my ten year old son Alexander, and my eight year old daughter Katherine. They have both given up many hours of play time, pool time, and story time so that daddy can do his homework. I am grateful for their unconditional love, and continual support. They are a delight and have been a constant source of energy and motivation.

I am very grateful to Alex Pelin for encouraging me nearly a decade ago to follow a line of research that I found really fascinating. His thoughtful advice and guidance were invaluable.

I would like to thank my major professor Tao Li for shepherding me through to the finish line with extreme passion. His boundless energy and enthusiasm have kept me motivated. I thank the committee members Ana Pasztor and Geoffrey Smith for their continuous support and encouragement. I also need to thank Raimund Ege for stepping in for a short but critical period of time. And I would like to thank Malek Adjouadi for committing his time and help in a pinch.

Many warm thanks go to Maria Monteagudo, Olga Carbonell and Martha Soledad who have been of invaluable help in dealing with the many details associated with my studies. I also want to express my deep gratitude to Roberto Hernandez for willingly falling into the trap of keeping me in line.

ABSTRACT OF THE DISSERTATION

THE STRUCTURE OF GAMES

by

David M. Kaiser

Florida International University, 2007

Miami, Florida

Professor Tao Li, Major Professor

Computer Game Playing has been an active area of research since Samuel's first Checkers player (Samuel 1959). Recently interest beyond the classic games of Chess and Checkers has led to competitions such as the General Game Playing competition, in which players have no beforehand knowledge of the games they are to play, and the Computer Poker Competition which force players to reason about imperfect information under conditions of uncertainty. The purpose of this dissertation is to explore the area of General Game Playing both specifically and generally.

On the specific side, we describe the design and implementation of our General Game Playing system OGRE. This system includes an innovative method for feature extraction that helped it to achieve second and fourth place in two international General Game Playing competitions.

On the more general side, we also introduce the Regular Game Language, which goes beyond current works to provide support for both stochastic and imperfect information games as well as the more traditional games.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

1. INTRODUCTION

Computer Game Playing is one of the oldest areas of endeavor in Artificial Intelligence. However, most AI research in the area of computer game playing has focused on a small number of very similar games (Halck 1999). Focusing on this limited category of games has had its advantages. The games are popular and simple enough that most people understand them without much explanation. They have proved complex enough to provide a real challenge in programming viable opponents. The popularity of these games has also served to generate interest in game-playing research.

While concentrating on these popular board games has born fruit, focusing on such a restricted problem space has certain disadvantages. For each new game, a large amount of the researcher's efforts must be spent to develop a program just to play the game. It is difficult to evaluate the research. The improvements in computer players can be the result of better hardware (more memory, more processors, faster processor, and better architecture), better evaluation functions, better code, or better tricks that are particular to a specific game. It is difficult to determine what components are applicable to other games or games in general. As Pell states: "We can write successful programs, even learning programs, without understanding the ability actually to analyze games, possible the most interesting issue in game-playing, from an AI perspective." (Pell 1993)

With champion-level computer players in Chess, Checkers, Othello, and Backgammon, Machine Learning research has extended into new and different games (Fürnkranz 2001). However these efforts are hampered by two things. First there is no generally recognized formalism for the structure of games. Second there is no popular grammar to define games. Researchers are forced to essentially start from scratch with

each new game. Because no generally acceptable formalism for game structure exists, current machine learning techniques used in game playing do not allow us to compare different kinds of games, or choose between viable learning alternatives, easily and cost-effectively.

In recent years research interests have extended beyond the classic board games into games such as Poker and Bridge. These games offer many new challenges because they are games of imperfect information, where decisions are made under conditions of uncertainty. More recently the area of General Game Playing has come into the spotlight. General Game Playing is concerned with developing systems to play arbitrary games for which they have no prior knowledge. The Game Description Language and General Game Playing framework developed at Stanford University (Genesereth, Love and Pell 2005) has been used in the first three AAAI General Game Playing competitions.

This dissertation has two main purposes. First this dissertation describes the design and implementation of a General Game Playing system, whose performance led to second and fourth place in two international competitions. The system implements several well-studied methods of computer game playing such as min-max with Alpha-Beta pruning, paranoid search and transposition tables. A high-level description of the general system design and the underlying principles is followed by a detailed explanation of a number of key implementation techniques and an innovative method for feature extraction.

Feature Extraction is an important problem in many areas of computer science including machine learning, data mining, computer vision, bioinformatics and speech recognition (Guyon and Elisseeff 2006). Our main contribution in this area is an original

technique for automatically identifying critical game features by examination of the game description and through self play.

Secondly this dissertation gives a proposal for a formalism that can be applied to all software development that use games as a domain. This formalism can then be used as the basis for machine learning in games to provide comparisons between the games and to explore the effects of variations on learning techniques. This language is capable of representing a wide range of games to facilitate research in Machine Learning and other areas of Artificial Intelligence.

The Regular Game Language is significant because unlike other languages for describing games, RGL is capable of representing both deterministic and non-deterministic games. Additionally, RGL is capable of representing games of both perfect and imperfect information. This makes RGL an ideal framework for researchers in Artificial Intelligence to explore many new areas of Computer Game Playing.

The dissertation is separated into two parts. The first part is based primarily on the design and implementation of our successful General Game Playing system (Kaiser 2007a) and our innovative method of automatic feature extraction for autonomous agents (Kaiser 2007b). The second part of the dissertation focuses on the design of a new framework for describing games. Originally outlined in (Kaiser 2005), the Regular Game Language allows the specification of both stochastic and imperfect information games.

2. INTRODUCTION TO PART I

This section of the dissertation introduces the subject of Computer Game Playing, both from the perspective of specialized game playing systems like Deep Blue and then from the perspective of General Game Playing. It then goes on to describe our particular implementation of a GGP system and the success we have had participating in international competitions.

Chapters 3 through 6 introduce background information and concepts that relate to the construction of a General Game Playing systems. Chapter 3 introduces concepts related to traditional Computer Game Playing. Chapter 4 discusses General Game Playing and describes the language used in the three AAAI General Game Playing competitions. Chapter 5 introduces the subject of feature extraction, which is closely related to one of our main contributions. Chapter 6 presents Automatic Theorem Proving which will help the reader in the sections discussing the design of OGRE, our General Game Playing system.

Chapters 7 through 10 describe our particular implementation of a General Game Playing system: OGRE. Chapter 7 outlines the basic architecture of the system. Chapter 8 details our innovative method for extracting game related features. Chapter 9 describes the construction of the evaluation function. Chapter 10 discusses how our system performed during international competition.

3. COMPUTER GAME PLAYING

Computer Game Playing has been an active area of computer research since Shannon's ground breaking paper outlined his strategy for programming a computer to play Chess (Shannon 1950). This section gives a brief overview of some terminology and concepts that have been used in computer game playing systems from the very earliest Checkers player (Samuel 1959).

3.1 Game Related Concepts

We define a *game* as a decision problem with one or more decision makers – players – where the outcome for each player may depend on the decisions made by all players. *One-person* games are puzzles, where a single player makes all the decisions. Several *two-person* games are mentioned in this dissertation. Chess, Checkers and Tic-Tac-Toe are examples of two-person games. In such games, there are two participants that are the decision makers. We will use the term *multi-player* games to refer to games with more than two players such as Chinese Checkers, or Bridge.

A game is called a *perfect information* game if all the players have complete information of the current game state. Othello is a perfect information game, because the state of the game is completely captured by the position of the pieces on the board, and all players have access to this information. Games in which players are not privy to the entire game state, such as Poker or Battleship, are *imperfect information* games.

A *deterministic* game is one in which the outcome of each move, or transition from one state in the game to another, is known beforehand. Checkers is a deterministic game. Both players know exactly what the results will be for any particular move. The opposite of a deterministic game is a *stochastic* game. Backgammon is an example of a

stochastic game. Stochastic games involve some element of chance such as rolling dice, spinning a wheel, or shuffling cards.

A *zero-sum* game is a game in which one player's winnings equal the other player's losses. If we add up the wins and losses in a game, treating losses as negatives, and we find that the sum is zero for each set of strategies chosen, then the game is a zero-sum game. In games that are not zero-sum games, the winnings of one player are not necessarily the losses of the other. Both players may win or both players may lose. A Poker game in which 20% of each player's winnings must go to the local charity is not a zero-sum game. Zero-sum games eliminate any incentive for cooperation between players. Many familiar two-player games like Chess, Checkers and Tic-Tac-Toe are zero-sum games.

3.2 Game Tree

A game can be represented as a *game tree*. A game tree is a directed graph that represents state space of a game. In deterministic games, each *node* in the tree represents a state in the game, each *edge* represents a move. The *root* of the tree is the initial state of the game, before any players have made any moves. A *terminal state* is a position where the rules determine when the game ends. A *terminal node* in the game tree represents a terminal state in the game (Bratko 1990). In stochastic games, chance nodes must be introduced to represent the variable elements of the game, such as dice rolls in Backgammon.

A node is *expanded* by generating all successors of the position represented by the node. A direct successor of a node is termed a *child* of the node. The direct predecessor of a node is termed the *parent* of the node. The root, or initial game state, is the only

node without a parent. Likewise *terminal nodes* are the only nodes without child nodes. A game tree is generated by expanding every node, starting at the root node, until all the terminal nodes are reached. Each terminal node has an associated value known as the *utility* value for the player. The partial game tree for Tic-Tac-Toe is shown in Figure 1 has a utility value of 1 for a win for player X, -1 for a loss and 0 for a draw.

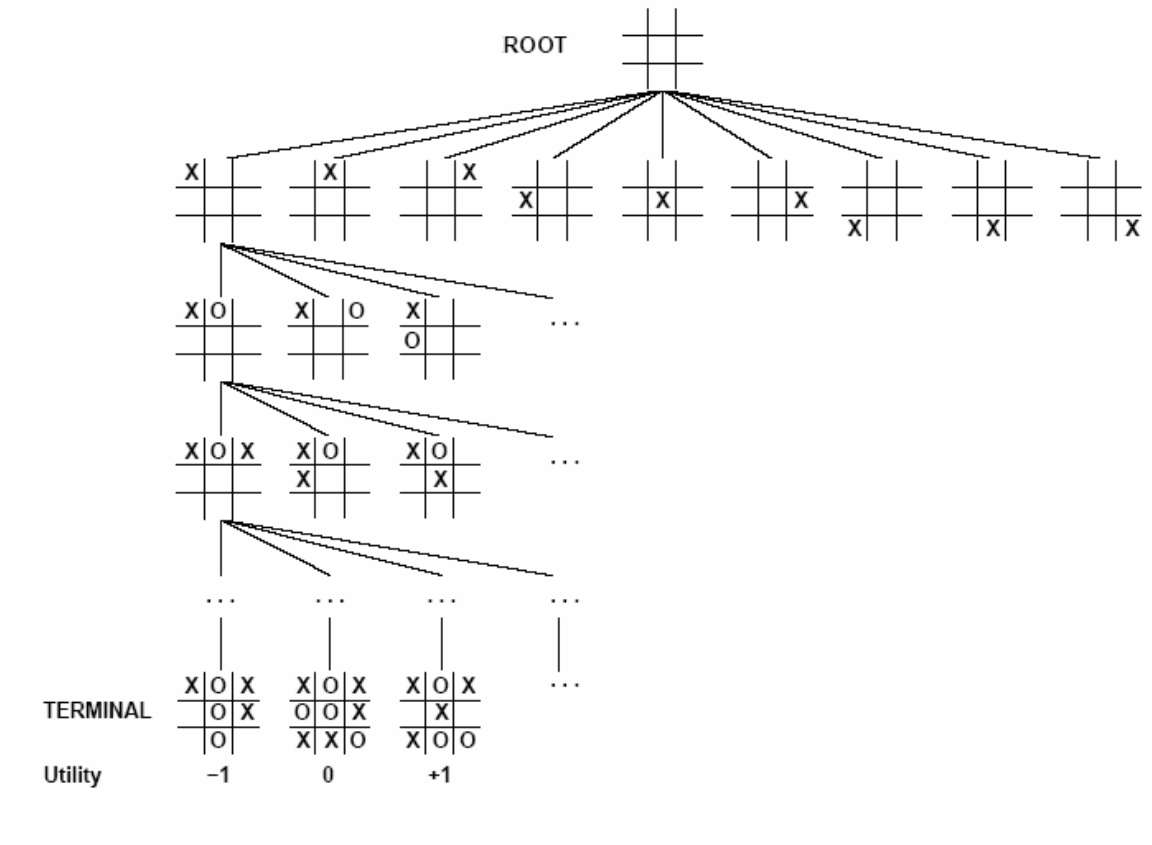

Figure 1. Partial game tree for Tic-Tac-Toe.

By examining the entire game tree of any particular game, from the root all the way down to the terminal nodes, it would be possible to find the best strategy for playing that game. However, for most interesting games the game tree is extremely large, and infeasible to generate completely. For example, the size of the game tree for Chess has been estimated as being between 10^{120} (Shannon 1950) and 10^{123} (Allis 1994) nodes. To

get an idea of how difficult it would be to generate this game tree, consider a computer capable of generating one billion (10^9) nodes every second. Such a hypothetical computer, if run continuously for five billion years, would only be capable of generating 10^{26} nodes in that span of time.

3.3 Search

When the game tree is too big to be generated in its entirety, a *search tree* is generated instead. The search tree is only a part of the game tree. The root of the search tree represents the game state under investigation. The search tree is generated during the search process. Nodes which have not been expanded yet are called *leaves*. When terminal nodes are not reached the utility value is computed for each game position by means of an *evaluation function*. The evaluation function produces an estimated utility value that indicates how good it would be for a player to reach that position.

Standard game-playing search techniques include some variant of the *min-max* algorithm with *Alpha-Beta pruning* (Levy and Newborn 1991). The concept is based on the observation that in some cases, it is clear that further investigation of part of the game tree is pointless. The basic principle is to expand a game tree from the current position, and evaluate each game state based on a heuristic evaluation function, pruning huge areas of the search space that appear unpromising. The Alpha-Beta algorithm has proven to be a valuable tool for the design of two-player, zero-sum, deterministic games with perfect information and there have been numerous algorithmic enhancements to improve the search efficiency.

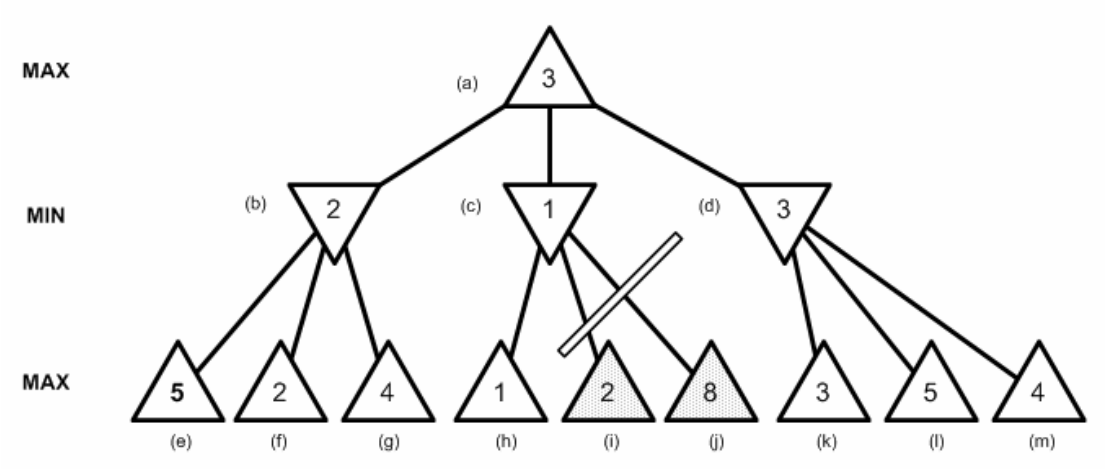

Figure 2. Using Alpha-Beta pruning nodes (i) and (j) do not need to be expanded.

In the basic min-max algorithm, player *max* is trying to maximize the utility of a position while player *min* is trying for a minimum value. An illustration of Alpha-Beta pruning is shown in Figure 2. The triangles pointing up are player *max*, while the triangles pointing down are player *min*. At node (b) player *min* will chose a move that leads to a position with the lowest utility, in this case node (f) with a utility value of 2. At node (c) player min would choose a move leading to (h) with a utility of 1. At this point, player max would have no further interest in this line of play, because it has a lower utility than the variation including node (b) so the search does not need to expand nodes (i) and (j).

The min-max algorithm was designed for two-player games, but many games such Chinese Checkers have multiple players. The *paranoid algorithm* (Sturtevant and Korf 2000) reduces an *n*-player game to a 2-player game by assuming that *n*-1 players have formed a coalition against the remaining player. In practice, this is unlikely, but the assumption makes it possible to implement the basic min-max with Alpha-Beta pruning algorithm with only minor modification.

3.4 Iterative Deepening

Iterative deepening (Russel and Norvig 2003) allows the system to examine all available moves in a reasonable amount of time. The idea is to search to a fixed depth in the search tree. If no winning variation is discovered, and we have some time left, then increase the depth of the search and try it again.

Figure 3 shows the basic algorithm.

```
for (depth = 1;; depth++) {
    value = AlphaBeta(node, depth);
    if (isTimeOut())
         break;
}
```

Figure 3. Basic iterative deepening algorithm.

Iterative deepening provides a simple means of interrupting the search when the amount of time runs out and it makes searching more efficient. The obvious disadvantage of this approach is that it re-visits the nodes at the top of the search tree, possibly many times. From a theoretical perspective this is not a serious drawback because the cost of expanding nodes at the lower depths dominates searching the upper nodes multiple times (Korf, Reid, and Edelkamp 2001). For a game tree with an *average branching factor* (average number of children expanded) of b, searching to depth of d has a time complexity of $O(b^d)$ which is equivalent to depth first search. From a practical perspective, these repeated expansions can also be addressed by using a *transposition table*.

3.5 Transposition Tables

In many games it is possible to reach a given position in more than one way and these are known as *transpositions*. For example there are four different sequences of moves that lead to the position in Figure 4, namely (a1, a2, a3, b1), (a1, b2, a3, a2), (a3, a2, a1, b1) and finally (a3, b1, a1, a2). One way to speed up the search process is to avoid evaluating the same position more than one time.

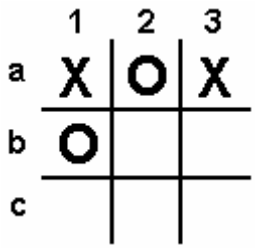

Figure 4. Four move variations can lead to this position in Tic-Tac-Toe.

The idea behind a *transposition table* is to store the evaluation value for evaluated positions into a table (Greenblatt, Eastlake, and Crocker, 1967). The entries in the table include not only the evaluation for the position but also a measure of how deep the search was which produced the value. Before a position is considered for evaluation, the table is consulted. If the position has previously been evaluated, and is found in the table, then the program can avoid further expansion on the node in question.

To maximize speed, the table itself is implemented as a *hash table (*Knuth 1973). Each game state is converted to a large number using a hashing function. A popular method used in many Chess programs is described by Zobrist (1970). When combined with iterative deepening a transposition table can dramatically speed up search. The

transposition table reduces the overhead of the iterative deepening algorithm by storing the evaluation results of previously visited game states.

4. GENERAL GAME PLAYING

General Game Playing (GGP) is the problem of designing systems that are capable of playing many different games successfully. Many specialized game playing systems are capable of beating the best human players in games such as Chess, Checkers, Othello and Backgammon. The most famous of these is Deep Blue which was the first program to defeat a reigning world Chess champion. Specialized game playing systems are designed for particular games and human expert knowledge is a key component of their success. GGP systems, on the other hand, must be able to play previously unknown games, based exclusively on the description of the game. Rather than relying on algorithms tuned for a specific game, GGP systems must be able to adapt their behavior to each new game they encounter.

To compete effectively, game playing agents must make a series of moves that lead to a final winning position. They search through the game tree assessing positions based on an evaluation function. To perform well, the evaluation function must be as accurate as possible. Systems designed to play specific games use optimization techniques such as opening books or end game databases to enhance the evaluation function. The World Chess Champion Deep Blue (Campbell, Hoane and Hsu 2002) has an opening book of 4,000 positions and a summary of 700,000 grandmaster games. World Checkers Champion Chinook (Schaeffer, Treloar, Lu and Bryant 1996) has perfect information for over 443 billion end game positions. The effectiveness of the evaluation function directly impacts the search for good moves. An accurate evaluation function allows the system to spend more time on promising areas of play and less time on obviously bad moves.

Successful GGP systems, however, must be capable of playing many different games, even games they have never seen before. The central problem for a GGP system is to construct a heuristic evaluation function that performs efficiently for each different game it confronts. Even if the system has access to a set of perfect evaluation functions for specific games, it must still determine whether or not one or more of these functions are applicable to the current problem it is facing.

In order to perform well, a general game playing agent must be able to examine the relevant features of different kinds of games and generate an evaluation function. The system must also accomplish all this within the limits set by the operational environment. These limits can include memory resources constraints, restrictions on the amount of time available to analyze the game definition and limits on the amount of time to make moves.

4.1 Game Description Language

The Game Description Language (GDL) along with the Stanford General Game Playing framework (Genesereth, Love, and Pell 2005) was used during the AAAI 2005 and 2006 General Game Playing Competitions. GDL is a formal language for defining deterministic games with perfect information.

In GDL, games are modeled as state machines in which the state of a game is described as a set of true facts at a specific point in time. The rules of the game are described using logical rules that define successor states in terms of the current state and player moves. The GDL is a variant of first-order logic that uses syntax from the Knowledge Interchange Format (KIF) language (Genesereth 1991).

```
1.   (role white)
2.   (init (cell 1 1 b))
3.   (<= (legal white (mark ?x ?y))
         (true (cell ?x ?y b)))
4.   (<= (next (cell ?m ?n x))
         (does white (mark ?m ?n))
         (true (cell ?m ?n b)))
5.   (<= (goal white 100)
         (true (cell 1 1 x)))
6.   (<= (terminal)
         (true (cell 1 1 x)))
```

Figure 5. A toy game description in GDL.

A toy game is described in Figure 5. Each GGP agent must be able to play any game, given such a description. In this example line 1 indicates the player `(role white)`. There is only one `role` statement and therefore only one player in this game. Line 2 `(init (cell 1 1 b))`, is the initial state of the game. At the start of the game one true fact will be set. A single location 1/1 will be set to blank. Again there is only one statement, making the layout of the game very simple. But a more complex game such as Chess, might have sixty-four such statements, one for each position on the board. After the start of the game, all `init` statements are converted to `true` statements.

Our toy example has one legal move `(legal white (mark 1 1))`. In this particular example the game is over when the player makes the first move and there is only one legal move. Using a theorem prover, a GGP agent can determine all the legal moves, game termination conditions, goal values and successive game states from the current game state.

This simple example contains all the key elements of game description in GDL, including the distinguished keywords: `role`, `init`, `true`, `legal`, `does`, `goal`, `terminal`. Tokens such as `cell` and `mark` are game specific and have no intrinsic meaning. Numbers are treated as symbols that have no significance outside that defined within the game description itself.

4.2 Survey of Previous General Game Playing Systems

Barney Pell introduced Meta-game playing (Pell 1993), a framework for general game playing systems to compete against each other in games for which they had no prior knowledge. Pell's work is a direct predecessor to the GDL and the Stanford Game Playing framework. Pell's METAGAMER program plays a class of games called symmetric Chess-like games (a subset of two-person, perfect information, deterministic, zero-sum games). The class includes the games of Chess, Tic-Tac-Toe, Checkers, and many others. METAGAMER supports only square boards (as in Chess and Checkers), but the board size can be changed. Also, the board can be defined as a cylinder so that the left and right sides are connected, allowing piece movement to wrap-around from one side of the board to the other. To allow for the ability to promote pieces (as in Chess or Checkers) a promotion rank can also be defined.

Pieces are defined in terms of moving, capturing and promoting, and by an optional set of constraints on the use of these abilities. For example, it is possible to define a piece that can capture opponent pieces by hopping over them diagonally (like a man in draughts), or a piece that can move indefinitely through a line of empty squares (like a rook in Chess). However, some types of movement cannot be represented: en passant or castling in Chess for example. With the Metagame grammar, it is possible to

define games that have most of the rules of many games, including Chess, Shogi, Checkers, Tic-Tac-Toe, and Go-Moku.

Gherrity (1993) created a program called SAL that has the ability to learn any two-player, deterministic, perfect information, zero-sum game. It does not learn from the rules, but by trial and error from actually playing and being given valid moves at each turn. The program consists of a game-independent kernel and a game-specific move generator module. The kernel remains unchanged for different games, while the move generator is modified to reflect the rules of each game to be played. The kernel uses a temporal difference procedure combined with a back propagation neural network to learn good evaluation functions for the game being played. The back propagation neural network is fully connected between layers, with a single layer of hidden units. The number of hidden units is one for every ten input units. Gherrity chose ten different feature types for the evaluation function. The actual number of features depends on the size of the board, and the number and types of pieces. For example, SAL calculates 61 features for Tic-Tac-Toe, resulting in a network with 61 inputs. SAL calculates 221 features for Connect-four, and 1031 for Chess. SAL has two evaluation functions, one for each player, so it can also learn non-symmetric games.

When it is SAL's turn to move, a 2-ply search tree is created. Further expansion is controlled by the consistency search procedure. This is a game-independent generalization of the standard Alpha-Beta search procedure. It is based on the idea that some positions may be evaluated incorrectly and it uses search to correct these evaluation errors (Gherrity 1993). SAL does not make use of a language to describe games. It

requires user-supplied move generators written in C, and some constants that declare the size of the board and the pieces used in the game.

Susan Epstein (1994) created a program called HOYLE that can learn to play two-person, perfect information, finite board games. It uses a mixture of generic and specific advisors weighted for each particular game to improve its performance. HOYLE has 23 game-independent advisors. Each advisor represents a different viewpoint on games playing, and takes a fairly narrow, but rational, view of the move selection problem. For example, one advisor may specialize in moves that can win from the current game state, while another only considers moves that eliminate the opponent's pieces. When it is HOYLE's turn to move, a procedure computes all legal moves from the current state and offers them to its advisors. The advisors comment on these alternatives, recommending them or advising against them, based on its narrow perspective. Based on the advisors' recommendations, a simple arithmetic vote selects a move to be made. HOYLE plays without ever searching more than 2-ply ahead in the game tree (Epstein, Gelfand, and Lesniak 1996).

HOYLE seems to be unique in that it improves its performance through a variety of learning paradigms. HOYLE begins with precise but general prior knowledge about the domain of two-person, perfect information, deterministic games. The program learns how to play a specific game gradually. The information it retains, or learns, from play is different for each advisor. One advisor may be interested in opening moves that have been made by expert (winning) opponents, while another attempts to store patterns credited for negative outcomes in play, yet another advisor may store no information at all (i.e. perform no learning). HOYLE quickly and efficiently identifies key information

about the game that is adequate for expert move selection. While Epstein's program learns much faster than SAL, it seems to require a certain amount of hand crafting (i.e. programmer intervention) for each game. HOYLE has learned to play Tic-Tac-Toe, Qubic and Nine-Men's Morris perfectly. It is unclear how well HOYLE would play complex games like Chess.

Robert Levinson developed MORPH II (Levinson 1994), a domain-independent machine learning system and problem solver. This is an extension of Morph (Gould and Levinson 1992), a program that learned to play Chess using Adaptive Predictive Search, a method by which search systems can improve through experience. The previous system, Morph, while given little initial domain knowledge, was able to learn to defeat human novices while searching only 1-ply. Like its predecessor, MORPH II also has a low reliance on search; just 2-ply is average. Games are presented to the system using a graph-theoretic representation scheme. Interestingly, Levinson states that games generated from the Metagame generators can fit this structure (Levinson 1995). In other words, MORPH II should be able to play Metagames.

Given the rules of a game, the MORPH II system is responsible for abstracting its own features and patterns, developing its own learning modules and adjusting the weights for state evaluation through training. The learning modules learn through both weight propagation (similar to a neural network) and pattern evolution (like genetic algorithms). While MORPH II has successfully learned to play Tic-Tac-Toe, it has yet to be tested in more difficult games.

WAR (Kaiser 2000) is a general game player designed to play a class of games called Simple War Games. The class includes both deterministic and non-deterministic

games that are comparable in complexity to Checkers and Chess. The language used to describe Simple War Games is not capable of expressing all deterministic games. The WAR system uses a genetic algorithm to learn each new game through self play. But WAR performs the learning process off-line, not as part of the game playing process.

MULTIGAME (Romein 2001) provides a language for describing single or two-player, deterministic, perfect information games. MULTIGAME also includes an environment for playing those games. The system is designed to provide fast, parallel search on a distributed memory system. It is structured as traditional game playing systems: a move generator, a search engine, an evaluation function, and heuristics that guide search. However, the system relies on the researcher to provide the evaluation function.

The most relevant work to our own efforts in both feature extraction and General Game Playing is that of Kuhlmann, Dresner and Stone (2005). The GGP system developed at the University of Texas competed in the first GGP competition. The system identifies certain structures that can be determined from the game description such as successor functions and also has an interesting method for identifying team-mates in multi-player games.

The method used by Kuhlman to identify movable pieces differs from our own. Their system hypothesizes game structures, such as boards and pieces, from the game description and then uses internal simulation to see if these hypotheses are violated. Our approach is statistical, while Kuhlmann's approach is based on the strongest unviolated constraint.

The commercial game Zillions of Games (Mallet and Lefler 2001) is a program capable of playing a whole host of different games. Rules are described by a language, similar to the programming language LISP, in which users can define different kinds of games including Chess, Checkers, Tic-Tac-Toe, Othello and many others. Over six hundred games have been written in the Zillions grammar. The language is capable of describing many deterministic, perfect information games, but it does not provide support for defining games in which more than a single piece occupies the same position simultaneously.

5. FEATURE EXTRACTION

Feature Extraction is an important problem in many areas of computer science including machine learning, data mining, computer vision, bioinformatics, and speech recognition (Guyon and Elisseeff 2006). Successful GGP agents must be able to extract pertinent features in each new game they are presented and adapt their behavior accordingly.

Feature identification is necessary whenever search cannot reach terminal game states from which to determine true payoffs. This is typical in most interesting games (Utgoff 2001). In a domain such as that considered here, namely GDL and the GGP competition, where the game definition is unknown beforehand, it is necessary for the system to have a way to identify useful measurable features and a method of combining these feature values into a single number that indicates the overall ranking of the game state relative to others. A GGP agent must be able to approximate the relative utility of the game states that it encounters in actual play or look-ahead search.

For many games, features such as pieces and their location relative to one another are critical to evaluating a game state. Other useful structures include turn counters, accumulators, game boards, successor functions and goal patterns. However the GDL has no formal mechanism for identifying game structures in the language itself. All such information is specific to each game definition. The game state from a game of Chess defined in GDL is used to illustrate our technique.

```
(true (cell a 1 wr))
(true (cell a 2 wp))
(true (cell a 3 b))
(true (cell a 4 b))
(true (cell a 5 b))
(true (cell a 6 b))
(true (cell a 7 bp))
(true (cell a 8 br))
(true (cell b 1 wn))
(true (cell b 2 wp))
(true (cell b 3 b))
.
.
.
(true (cell h 6 b))
(true (cell h 7 bp))
(true (cell h 8 br))
(true (control white))
(true (step 1))
```

Figure 6. Partial game state for Chess in GDL.

In the Stanford GGP framework, games are modeled as state machines, where a state is a set of true facts at a given time. A partial game state for the game of "Chess" is shown in Figure 6. In our Chess example the first and second arguments (i.e. `<arg1> <arg2>`) represent board locations while the third argument (i.e. `<arg3>`) represents the pieces. Each piece is represented by a two letter combination (e.g. "`wr`" for the white rook), and empty positions are indicated with the token "`b`".

```
Example: (true (cell a 1 wr))

Format: (true (<predicate> <arg1> <arg2> <arg3>))
```

But this format is neither necessary nor required. The tokens within game descriptions are normally obfuscated during competition. The tokens `cell`, `a`, `1`, and

`wr` might easily be presented to the GGP agent as `wearer`, `boling`, `undriese`, and `parfulds`. The GGP system cannot depend on the token strings to identify features.

Additionally, the arguments may not conform to any specific pattern. The order of the information could be scrambled or inverted, extraneous tokens might be added or the information may be combined in many different ways as shown in Figure 7. Example (a) is the original format. In example (b) the coordinates and piece have been reordered. Example (c) contains extraneous information, namely the symbol `dummy`. Finally (d) combines the coordinates into a single parameter.

Since the number of possible arrangements is limited only by the game author's imagination, it is worth while to develop a method to extract features that is more robust than simple pattern matching.

```
(true (cell a 1 wr))          /* a. original */
(true (cell wr 1 a))          /* b. reordered */
(true (cell dummy a wr 1))    /* c. noisy */
(true (cell a1 wr))           /* d. combined */

(true (place column1 row1 fortress))      /* a. */
(true (tyrant castle alpha prime))        /* b. */
(true (location noise primo keep first)) /* c. */
(true (position angrymuffin tower))       /* d. */
```

Figure 7. Alternate game structure formats, before and after obfuscation.

6. AUTOMATIC THEOREM PROVING

While not a topic typically associated with general game playing, Automatic Theorem Proving (ATP) is critical to processing GDL game descriptions. ATP deals with the development of computer programs to prove mathematical theorems. The language in which the theorems are written is a logic, often classical first-order logic. Theorems are composed of axioms and hypotheses which will lead (or won't) to a conjecture. ATP systems produce proofs that describe how and why the conjecture is a logical consequent of the axioms and hypotheses.

The proofs produced by ATP systems are intended to be readily understandable. For example, if the Towers of Hanoi puzzle were formulated as a theorem, the proof would describe the sequence of moves that need to be made in order to solve the puzzle.

ATP systems have been successfully used in many fields including software generation, software verification, security protocol verification and hardware verification. There are several ATP systems available. Some well-known and successful first-order logic systems are Otter, E, SPASS, Vampire and Waldmeister (Sutcliffe, Fuchs and Suttner 2000).

6.1 Inference

In logic, a *rule of inference* is a pattern of reasoning consisting of one set of sentences, called *premises*, and a second set of sentences called *conclusions*. The following is a rule of inference called Modus Ponens.

```
A -> B    if A is true then B is true
A         A is true
-------   therefore
B         B is true
```

ATP systems use a wide variety of inference strategies such unit resolution, linear resolution, set of support resolution, and term rewriting. The inference system Vampire uses ordered binary resolution, superposition, and splitting (Voronkov 1994).

To prove a theorem, the axioms of the theory to be proved are first put into a normal form called clausal form. An inference algorithm is then applied exhaustively to the resulting set of clauses in the search for a contradiction (the empty clause). The core component that performs the inference algorithm is sometimes referred to as the *inference engine*.

First-order theorem provers such as Vampire use saturation algorithms (Riazanov and Voronkov 2003). For each new clause generated by an inference the prover decides whether this clause should be kept or discarded. For non-trivial theorems a very large number of intermediate clauses need to be generated before the empty clause is found. This process can lead to a combinatorial explosion, so most systems perform inferences not on all kept clauses but only on a subset of them.

ATP systems essentially explore a tree of clauses, generated by their rules of inference, searching for the empty clause. But as is the case with game trees, generating a complete search tree can exhaust the available computational resources. Therefore, like game playing systems, ATP systems use heuristics (i.e. rules of thumb) for pruning inference steps and for guiding the search through the space of inference steps.

In the E Equational Theorem Prover, search control heuristics define the order in which the prover considers newly-generated clauses. A heuristic is defined by a set of clause evaluation functions and a selection scheme which defines how many clauses are selected according to each evaluation function (Schulz 2001).

The ATP system known as Gandalf is able to adapt its behavior. The system automatically selects search strategies that are likely to be useful for a given problem (Tammel 1997). This is exactly the kind of behavior one would desire in a general game playing system.

In simple rule-based inference engines, there are two methods of reasoning, *forward chaining* and *backward chaining*. Forward chaining applies a rule of inference (for example modus ponens) to the available facts, generating new facts in an attempt to reach a goal. Backward chaining starts with a list of goals (or hypothesis) and works backwards to see if there are facts available to support any of the goals. Programming languages such as Prolog support backward chaining.

Comparisons between systems are based mostly on success rates during timed competition using standard collections of problems. The main collection of problems is the Thousands of Problems for Theorem Provers (TPTP) library (Sutcliffe and Suttner 1998). This is used as the basis for the annual CADE ATP System Competition (CASC), held at the Conference on Automated Deduction (CADE) (Sutcliffe 2001).

7. OGRE

This chapter describes the implementation architecture and design issues behind OGRE, a General Game Playing agent. The system is fully implemented and competed successfully at the second international AAAI General Game Playing competition held at AAAI-2006. OGRE came in fourth place out of twelve initial participants. The OGRE system builds upon our success with GOBLIN, another GGP system that placed second out of seven competitors at the first international AAAI-2005 General Game Playing competition. The basic architecture of GOBLIN and OGRE are the same. The only major difference is that OGRE contains a new inference engine and our new feature extraction algorithm.

7.1 The Design

OGRE is a GGP agent designed to play any game defined in the Game Description Language (GDL) and to compete within the Stanford GGP framework. The Stanford GGP framework defines how participating agents compete. GGP agents are given the game description, their roles within the game, and the time limits available to analyze the game and the time available to submit moves.

The system consists of five key components: HTTP Interface, Parser, Game Analyzer, Search Engine and Inference Engine. The HTTP Interface controls all communication with the outside world. The Parser converts GDL into an internal clausal form suitable for the Inference Engine. The Inference Engine process clauses and is used to determine legal moves, successor game states, terminal conditions and goal conditions.

The overall process architecture of the OGRE implementation is shown in Figure 8. The Game Analyzer component is consulted only during the analysis phase, before the first turn is submitted.

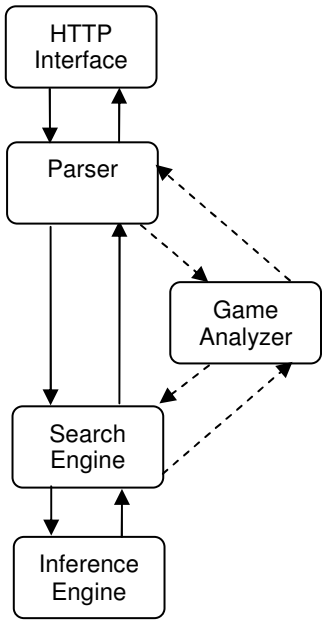

Figure 8. OGRE architecture.

7.2 HTTP Interface

The OGRE HTTP interface is a very simple HTTP server. The Stanford GGP framework (Genesereth, Love and Pell 2005) requires that each player communicates with the Game Master through an HTTP connection. The Game Master transmits all game information to the players including the game description, start time, player moves and final game scores. Players communicate only with the Game Master, sending legal moves at the appropriate time.

7.3 Parser

Game descriptions and player moves are extracted from the Game Master messages and sent to the Parser. OGRE uses the KIF parser built into the Java Theorem Prover (JTP) to convert the game description and player moves into clauses. JTP is a inference engine developed at Stanford (Fikes, Frank, and Jenkins 2003). OGRE's ancestor, GOBLIN, used the JTP inference engine not only for parsing but also to determine game states, legal moves, goal states, and game termination conditions. OGRE uses an entirely new inference engine. The new inference engine is significantly faster than the JTP inference engine and includes several GDL-specific enhancements.

7.4 Game Analyzer

There are two distinct phases of each match, the start phase and the play phase. The Stanford GGP process gives agents a period of time to analyze the game before the first turn begins. This start time can range from as short as a few seconds, to as long as an hour. Prior to the first message from the Game Master, agents have no knowledge of the game rules or the amount of time they will have to deliberate between moves. The OGRE system uses approximately 50% of the time given in the analysis phase to extract features and construct an evaluation function by playing games internally against a player that makes random moves.

OGRE attempts to generate an efficient evaluation function. It does so by examining the syntactic structure of the game definition as well as dynamic features that appear in the game during a self play stage. Features recognized solely from the game definition include the dependency graph, static predicates, successor functions, and turn counters. Features discovered through self play include pieces and board position. Our

innovative method for feature extraction is described in chapter 8. Construction of the evaluation function is described more fully in chapter 9.

Finally with the remaining time in the analysis phase, OGRE attempts to choose the best first move using the generated evaluation function. After the first move of the game is made, the system no longer references the Game Analyzer.

7.5 Search Engine

Since GDL allows multiplayer games, OGRE uses a variant of min-max with Alpha-Beta pruning called the paranoid algorithm (Sturtevant and Korf 2000). This essentially assumes that all of the opponents have formed a coalition and work together against the agent. This eventuality is highly unlikely, especially in the context of the AAAI GGP competition, where communication between systems in not possible, but the assumption reduces an n-player game to a two-player game, making it possible to implement the basic min-max with Alpha-Beta pruning algorithm with only minor modification.

Two common enhancements, iterative deepening and a transposition table (Sakuta and Iida 2000) are also used to improve performance of the search algorithm. Iterative deepening allows the system to examine all available moves in a reasonable amount of time. The transposition table reduces the overhead of the iterative deepening algorithm by storing the evaluation results of previously visited game states.

During development we found that playing certain puzzles within a reasonable amount of time is completely impossible without a transposition table. For the transposition table to operate effectively, however, it is absolutely crucial that the system identifies potentially misleading game state information. As noted previously, many

game definitions include elements such as turn counters. These elements must not be included in the game state hashing function or the benefits of the transposition table will be lost.

7.6 Inference Engine

Every state of the game, and every game state visited in the game tree must be interpreted by the inference engine. Depending on the game definition, OGRE can spend upwards of 71% of its time doing inferences. At the high end of the spectrum, in games that have complex definitions, this seriously limits how much time the system can spend on analyzing the game structure, as well as how deeply the system can search the game tree.

As stated previously, OGRE uses an inference engine that is significantly faster than the one used by its predecessor GOBLIN. The inference engine includes some enhancements specifically designed to improve performance within games defined in GDL. The basic inference algorithm is shown in Figure 9. The `queryStatic()` function stores and retrieves literals from the cache.

The most significant enhancement is the s*tatic predicate* cache. *Static predicates* are those predicates that are not dependent on the reserved GDL predicates `TRUE` or `DOES`. The `TRUE` predicates are facts that represent the state of the game. The `DOES` predicates are facts representing the moves made by each player. Any predicate that is dependent on either the `TRUE` or `DOES` predicate may need to be recalculated each turn. However, certain predicates will never change their results; therefore performing resolution on these predicates more than one time is wasteful. The OGRE inference engine caches the results of any resolution done on a clause containing a s*tatic predicate*.

```
Solve()
WHILE (TRUE) {
    IF goal stack G is empty
    THEN return TRUE
    goal G1 <- top literal in G
    IF out of time
    THEN return FALSE
    IF term of G1 is a static predicate
        AND cache contains G1 term
    THEN R <- queryStatic(G1)
    ELSE  R <- literals potentially
        unifiable with compliment of G1.
    ENDIF
    FOR each literal L in R
        IF L and G1 unify with mgu θ THEN
            G2 = Unify(L,G1, θ)
            Push right-literals of G2 onto G
        ELSE
            IF backtrack() fails
            THEN return FALSE
        ENDIF
    ENDFOR
 }
```

Figure 9. Inference Algorithm.

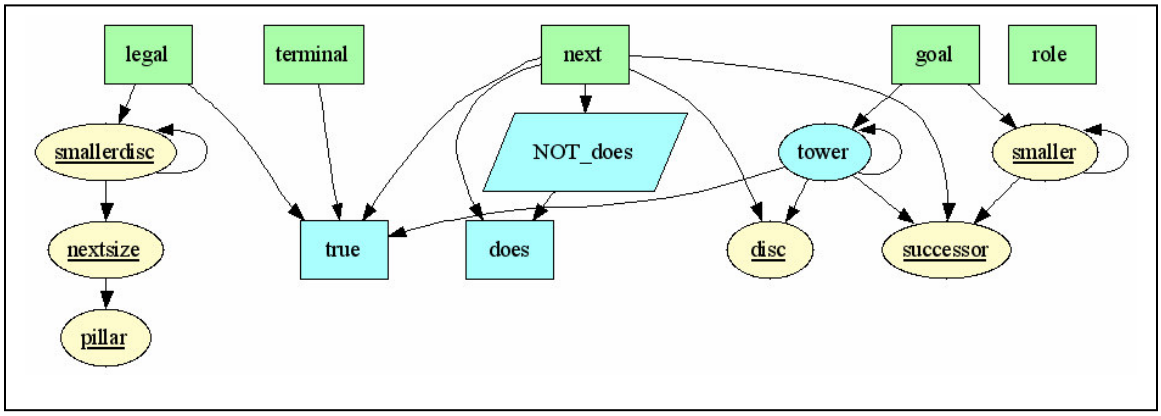

Figure 10. Dependency graph for predicates in the game Towers of Hanoi.

7.7 Identifying Static Clauses

GDL contains seven reserved predicates: `LEGAL`, `TERMINAL`, `NEXT`, `GOAL`, `ROLE`, `DOES` and `TRUE`. Any clause that is dependent on the reserved predicates `TRUE` or `DOES` is a *dynamic* clause. *Dynamic* clauses can change from turn to turn and therefore must be continually reevaluated. A clause that is not dependent on a `TRUE` or `DOES` predicate is a *static* clause. *Static* clauses need to be resolved only one time and are used to optimize the inference engine for the target game.

OGRE creates a dependency graph of the predicates in the game description. The dependency graph identifies clauses that are static. Figure 10 shows a dependency graph generated by the system. Rectangles represent reserved GDL predicates. Ovals represent game-specific predicates. Parallelograms indicate negated predicates. Static predicates are underlined. In this example, the predicate `smallerdisc` would only need to be resolved one time for a given set of parameters, because it does not rely on any `TRUE` or `DOES` predicates.

The *static predicate* cache can improve the performance of the inference engine significantly. That is to say that the system can do more inferences in the same amount of time. However, this improvement is heavily dependent on the game definition. Game definitions that make wide use of static predicates will benefit more than game definitions which have no static predicates or rarely use them. Another concern is that the amount of overhead necessary to maintain the cache can cost more in time than it saves.

7.8 Chapter Summary

In this chapter, we have given an overview of the design and implementation of a fully implemented autonomous General Game Playing agent called OGRE. The system automatically generates evaluation functions from game descriptions given in the Game Description Language. The effectiveness of this process allowed the system to come in fourth place in the 2006 AAAI General Game Playing competition.

8. EXTRACTING FEATURES

One feature that the system attempts to identify is the turn counter. GDL games are guaranteed to end in a finite number of turns. Many GDL games achieve this by using a turn counter. These turn counters are particularly vexing because game states that might otherwise be identical appear unique when there is a turn counter. Take for example, two instances of the eight puzzle game state:

```
a) (true (puzzle 1 b 2 3 4 5 6 7 8)) (true (turn 14))
b) (true (puzzle 1 b 2 3 4 5 6 7 8)) (true (turn 19))
```

By identifying the turn counter, our agent is able to recognize that these two game states are essentially the same, except that they occur at different times. Without this feature, all the benefits of the transposition table are lost. Puzzles that contain a turn counters become unsolvable if the information is not dealt with properly.

In order to identify a turn counter it is necessary to first recognize successor functions. Any series of predicate functions with the following format are considered possible candidates for successor functions:

(<successor> $\langle value_0 \rangle$ $\langle value_1 \rangle$)

(<successor> $\langle value_1 \rangle$ $\langle value_2 \rangle$)

(<successor> $\langle value_2 \rangle$ $\langle value_3 \rangle$)

. . .

(<successor> $\langle value_{N-1} \rangle$ $\langle value_N \rangle$)

Where <successor> can be any relation constant, and the <value$_n$> components are object constants. It is then possible to identify predicate functions with the following format as possible turn counters.

```
(<== (NEXT (<turn> <varY>))
     (TRUE (<turn> <varX>))
       (<successor> <varX> <varY>))
```

It is interesting to note that our solution for this problem is quite similar to that described in (Kuhlmann, Dresner and Stone 2005). This is likely an artifact of the sample game descriptions that were available during the development of these systems. The method, however, is quite brittle. Encoding the turn counter differently prevents recognition of this feature.

8.1 Extracting Features by Variance

In most games, players alter the game state by moving or placing pieces on a board. The basic idea of our approach is to compare facts from one turn to another and identify arguments in which symbols change. Arguments that represent these pieces will tend to have many changes, while arguments that represent static information like location coordinates will tend to remain the same. There are several steps to this identification process. First, groups of similar facts must be identified. Second, a consistent sorting scheme is determined for each fact group. Third, fact groups from sequential game states are compared to identify arguments that have changed.

In order to make comparisons possible, the system first categorizes the facts of the game state into separate groups based on the first predicate and the number of arguments. Thus the Chess example has three initial fact groups: `cell/3`, `control/1` and `step/1`.

8.2 Sorting By Variance

Like other first-order logic based calculi the GDL does not explicitly indicate the order that transitions are performed. There is no guarantee that the facts of each game state will be in any particular order. Therefore it is necessary to sort the members of each fact group in a consistent manner. Sorting is done based on the symbols in each argument of the fact predicate (i.e. `(true (<pred> <arg1> <arg2> ... <arg`*n*`>))`). But instead of sorting on the original order of the arguments, the system sorts the facts based on the variance of the arguments during the course of a game. The arguments with the lowest variance are sorted first. The understanding being that the variance for arguments representing a fixed grid will be zero, while the variance for highly mobile pieces will be quite large.

The variance is calculated separately for each argument position in each fact group. The standard formula for calculating an unbiased estimate of the population variance *s* from a finite sample of *n* observations is:

$$s^2 = \frac{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n - 1}$$

However, the numerically stable algorithm in Figure 11, due to Knuth (1988), who cites Welford (1962), is used because it does not require storing the whole sequence

of elements. Only the last value of the mean and the sum of squares are required. While the current system does not take advantage of this, it is expected that the ability to calculate this information on-the-fly will be of great benefit to future autonomous GGP agents.

```
Calculate Variance
For Every x;
  n = n + 1;
  delta = x - mean ;
  mean = mean  + delta / n;
  sumsqr = sumsqr + delta * (x - mean);
  variance = sumsqr/(n + 1);
End For;
```

Figure 11. Pseudo code to calculate variance.

The variance and mean are calculated separately for every argument in each fact group. The variable *x* is a unique number assigned to each symbol in the game description. The variable *n* is the total number of symbols processed by the algorithm.

After playing a few random games, the sort sequence for the Chess example is determined as shown in Table 1. Three new fact groups appear in the table. These were not part of the initial game state but identified as a result of the self-play process. `Pawn` represents en passant conditions, `Moved` tracks castling conditions and `Check` indicates the check state. The system identifies the third argument of `Cell` as having a high variance. The sort sequence mirrors the original arguments and so is unremarkable, but for the `Moved` predicate it can be seen that the arguments are reordered based on the ranking of their variance.

Table 1. Variances calculated from random game.

	Args	N	Mean	Col. Var.	Sort Sequence
Cell/3	arg1	592	8	0	1
Cell/3	arg2	592	8	0	2
Cell/3	arg3	592	5.14	92.50	3
Control/1	arg1	74	1	0	1
Step/1	arg1	74	1	0	1
Pawn/1	arg1	6	1	0	1
Pawn/1	arg2	6	1	0	2
Moved/3	arg1	103	1.09	0.08	1
Moved/3	arg2	103	1.64	0.22	3
Moved/3	arg3	103	1.23	0.17	2
Check/4	arg1	2	1	0	1
Check/4	arg2	2	1	0	2
Check/4	arg3	2	1	0	3
Check/4	arg4	2	1	0	4

For games with stable board locations like Chess and Checkers the arguments representing these board locations will have a very low variance since the available positions never change; there are always sixty four squares on a Chess board. Similarly as pieces are placed on the board, in games like Othello or Tic−Tac−Toe, or are captured, as in Checkers and Chess, the variance will increase.

In some games, like Chinese Checkers, the pieces move around but are never removed from the game. In these cases the variance will tend be much smaller and thus be of less usefulness in identifying pieces. Further measures are necessary to identify possible pieces and board locations in these types of games.

8.3 Motion Detection

After the fact groups are sorted in a consistent manner, the system attempts to determine which symbols are moving around. The motion detection algorithm performs a comparison operation on two sequential game states. Symbols that appear in the same

location in both states are ignored. Symbols that change or "move" are identified and retained.

Continuing with our Chess example, performing the comparison operation on two sequential game states would produce results shown in Figure 12, after white has moved a pawn from a,2 to a,3. Four things have changed. The turn counter step has been incremented; black is now the player to move; location a,2 is empty (or blank) and location a,3 contains symbol wp (for white pawn).

```
(step 1)         vs (step 2)         => (. 2)
(control white)  vs (control black)  => (. black)
(cell a 1 wr)    vs (cell a 1 wr)    => (. . . .)
(cell a 2 wp)    vs (cell a 2 b)     => (. . . b)
(cell a 3 b)     vs (cell a 3 wp)    => (. . . wp)

/* all others resolve to (. . . .) */
```

Figure 12. Example of comparing two game states

8.4 Piece Identification

The algorithm for extracting "piece" features from the game description is shown in Figure 13. The system first plays several random games and stores the results. The remaining calculations are all based on this recorded history.

After the fact groups have been identified, variances calculated and the sort order determined, the system begins comparing fact groups from successive game states. Each game history *H* contains k game states. Each game state s_k contains *m* fact groups. Each fact group g_i is compared against its succeeding g_{i+1} game state. Arguments in each fact group with a high variance have already been identified to create the sort order. The comparison operation identifies which symbols in these columns move, and with what frequency.

```
Play m Random Games, store game history H
Identify Fact Groups G
For every fact group g in G
  For every argument a in g
    For every symbols x in a
      Calculate variance v of x in a
    End For
  End For
  Determine sort order for g
End For
For each game state s in game history H
  For each fact group g_i in s
    Sort g_i
    /* Detect motion */
    Compare g_i with g_i+1
  End For
End For
```

Figure 13. Algorithm for extracting game "piece" features.

```
PieceInfo:
 symid=10, symbol=wr, argument=3, symbolvariance=0.134054, symbolmean=2.162162.
 symid=12, symbol=wp, argument=3, symbolvariance=3.127927, symbolmean=6.270270.
 symid=19, symbol=bp, argument=3, symbolvariance=1.042162, symbolmean=6.567565.
 symid=21, symbol=br, argument=3, symbolvariance=0.0,      symbolmean=2.0.
 symid=22, symbol=wn, argument=3, symbolvariance=0.243783, symbolmean=1.445945.
 symid=23, symbol=bn, argument=3, symbolvariance=0.0,      symbolmean=2.0.
 symid=25, symbol=wb, argument=3, symbolvariance=0.211351, symbolmean=1.689189.
 symid=26, symbol=bb, argument=3, symbolvariance=0.038378, symbolmean=2.040540.
 symid=28, symbol=wq, argument=3, symbolvariance=0.0,      symbolmean=1.0.
 symid=29, symbol=bq, argument=3, symbolvariance=0.0,      symbolmean=1.0.
 symid=31, symbol=wk, argument=3, symbolvariance=0.142882, symbolmean=1.175675.
 symid=32, symbol=bk, argument=3, symbolvariance=0.0,      symbolmean=1.0.
```

Figure 14. Features identified as pieces from the Chess game description.

Figure 14 shows an example of symbols identified as pieces from the Chess game example. The variable `symid` is the unique number assigned to each symbol while the `groupkey/argument` combination indicates where the symbol is used as a piece. This is necessary because the same symbol may have different meaning depending on the context. In our Chess example, the symbol "`b`" indicates a coordinate when it appears in

the first argument of `cell/3` but the same symbol indicates an empty, or blank, square when it appears in the third argument.

Symbols are associated with players by determining the available moves each turn and detecting what symbols moved. Symbols that appear to move regardless of player actions, such as the symbol "`b`" in our example, are not considered "pieces" under player control.

The entire procedure works best on games that explicitly define the entire game state on each turn. In practice, playing a few random games provides sufficient information to determine pieces and board locations for most applicable game definitions. However a more robust learning agent could be designed to explore the game tree more purposefully, trying new areas and expanding unvisited branches of the game tree.

8.5 Chapter Summary

This chapter introduced an innovative method for autonomous agents to extract key features from their knowledge of the environment. This method has been fully implemented in an autonomous agent that competed successfully in the second AAAI General Game Playing competition. The autonomous agent and how the extracted features are automatically incorporated into the final evaluation function are described in detail in the next chapter.

9. EVALUATION FUNCTION CONSTRUCTION

Following the approach used in HOYLE (Epstein 1994) and WAR (Kaiser 2000) we created several evaluators that encapsulate knowledge about common features found in many classes of games. This section explains briefly a partial list of evaluators implemented in OGRE. The evaluators can be categorized into two groups based on whether or not they rely on information derived from the structure of the game or simply the game definition itself.

9.1 Game Structure Evaluators

The first group of evaluators generalizes concepts related to games that involve boards and pieces. The complicating factor with these evaluators is that the GDL does not explicitly identify critical features such as pieces and board locations. In those cases where the system is unable to identify the required aspects of the game, these structural evaluators will not be available.

`Distance-Initial (Run-Away):` measures the distance between the initial position of a piece and the current position. This evaluator was intended for racing games like race-track-corridor and Chinese Checkers. Surprisingly it also provides a positive influence in games such as Checkers or Chess by nudging the agent into early board development.

`Distance-To-Target:` measures the distance between the piece and a target location. It is intended for games like Maze where a piece must be moved to a specific location.

`Count-Pieces:` measures the number of each type of piece in the current game state. This evaluator is most valuable in games where capturing pieces is possible, like Chess. Games like Tic-tac-toe do not benefit from it.

`Occupied-Columns:` This evaluator measures how many pieces are in the same column. This evaluator is intended to provide useful information for games like Tic-tac-toe, Pente, Connect-4 or the Eight Queens puzzle.

9.2 Game Definition Evaluators

Evaluators in the second group do not rely on information derived from the structure of the game. These evaluators encapsulate very general heuristics that are applicable to a broad set of games.

`Count-Moves:` measures the number of choices available to each player. In games such as Chess it can be beneficial to limit the choices available to the opponent.

`Depth:` produces a number inversely proportional to the search depth. The idea is to give a small preference for shorter solution paths. The evaluator is intended for puzzles which usually reward players for shorter solutions, but other games benefit as well. This evaluator completely ignores the game state.

`Exact:` calculates the exact value of the current game state based on the goal predicates given in the game definition. Depending on the game definition this evaluator will most often return a value at terminal game states. This function relies on the Inference Engine and this is therefore quite expensive.

`Pattern:` compares the current game state with a pattern found in the goal state. Helps solve several simple puzzles quickly.

`Purse:` measures the value of ordinal symbols in the game state. This evaluator is intended for games that involve accumulating items such as gold, chips or money.

9.3 Combining Evaluators

The system combines these evaluators into a single evaluation function by playing games internally against a player that makes random moves. The system uses approximately 50% of the time given in the analysis phase to both extract features and construct an evaluation function for the current game. To facilitate feature extraction, the agent plays two games in which all players choose their moves at random, and performs the feature extraction method described in section 7. This allows the system to quickly categorize structures that represent pieces and board locations.

The remaining portion of the self play stage is spent conducting a series of games in an effort to identify evaluators that are effective for the target game definition. For each unique piece type identified previously, an evaluator is created. Each of these evaluators is then used as the sole evaluation function in a quick game played against a player that makes random move selections. In order to play many games, the depth of look-ahead used in the search function is limited.

```
SelectEvaluators
    FOR each evaluator E
        FOR each piece P
            Create instance of E (En) using P.
            Play one game using En weighted +10.
            IF win THEN
                add En to list L
            ELSE
                Play game using En weighted -10.
            IF win THEN
                add En to list L
        ENDFOR (piece)
    ENDFOR (evaluator)
    RETURN list of evaluators L
```

Figure 15. Algorithm for selecting evaluators

Every evaluator returns a positive number, although each evaluator may further be modified by a positive or negative weight. The algorithm used to select evaluators which involve pieces is outlined in Figure 15. One instance of each evaluator is created for each piece identified in the game definition.

The final evaluation function is the sum of weighted values returned by all the selected evaluators as shown in the following formula where $e \in L$.

$$\sum_{i=1}^{n} e_i w_i$$

The actual values generated by the final evaluation function are unimportant. What does matter is that the function be able to give an assessment of the game state that is accurate relative to the other game states. Currently, the system assigns similar weights to each evaluator, but it might prove beneficial to test different weighting strategies.

10. RESULTS

The AAAI-06 competition consisted of a series of matches held over the course of three months, from May-June 2006. GGP agents participated remotely in the first three rounds of the competition and the competition culminated in a fourth and final round of matches at the conference in Boston.

Participants played a variety of games generated by the competition organizers. The games included single-player puzzles such as n-Queens, peg jumping, the Towers of Hanoi as well as planning and scheduling problems. Two player games included variants of Tic-tac-toe, Othello, Chess and Checkers. Multi-player games included variants of Chinese Checkers, and Othello.

Players had the opportunity to earn up to 100 points from each match they participated in. The points were explicitly defined in each game definition. Zero-sum games such as Tic-tac-toe allow only one player to gain 100 points. Some puzzles afforded the opportunity to gain less than the full number of points by partially completing the goal. Other games allowed ties or somehow distributed points between players. Several cooperative games organized players into teams, and each member of the winning team received 100 points.

The matches were organized into rounds. The points accumulated in each round were weighted; victories in later rounds were more advantageous than those in the earliest rounds. The matches in the first round were weighted 0.25, round two matches weighted at 0.50, round three 0.75, and round four matches were weighted at 1.00. Total scores of all previous matches were used to seed the contestants in the two-player

matches of round four. The two top-scoring agents competed in the final championship match at the conference.

10.1 Assessment

Our system, OGRE, performed successfully during the entire three months of the competition, coming in fourth place out of twelve initial entrants (Love 2006). OGRE played 41 different games including one-player games (puzzles), two-player games, and games involving three or more players. Some games were played more than once with players switching sides. As shown in Table 2, our system won 34% of the matches it participated in.

Table 2. Results for games participated in during the AAAI competition

	Results				Percent			
Games	Won	Partial	Loss	Total	Won	Partial	Loss	Total
Puzzles	5	6	8	19	26	31	42	100
Two Player	18	14	14	46	39	30	30	100
Multi	2	3	3	8	25	37	37	100
Total	25	23	25	73	34	31	34	100

Our agent was designed primarily to play two player games and this bias is apparent in the game statistics. The system performed better in two-player games than it did in puzzles.

Table 3. The un-weighted points acquired by the top four players.

		One-player Points		Two-player Points		Multi player Points	
Fluxplayer	1st	1520	80%	2792	59%	350	50%
Cluneplayer	2nd	1145	60%	2895	62%	300	43%
Pires5600	3rd	1000	53%	2923	62%	200	29%
OGRE	4th	825	43%	2322	49%	450	64%
Total Possible Points		1900		4700		700	

Each match afforded players the opportunity to acquire up to 100 points and each participant had the opportunity to earn up to 7300 un-weighted points. Table 3 shows the total number of unweighted points acquired by the top four players. OGRE received only 43% of the possible points from puzzles compared to the 1st place finisher Fluxplayer, which receive 80%. On the other hand OGRE did better than the top three players in games with more than two players, winning 64% of the possible points in this category.

OGRE finished in 4th place while its predecessor, GOBLIN, had finished 2nd the previous year. All of the top four finishers had participated in the first GGP competition and demonstrated clear improvements in performance.

On average our system is only capable of searching 100 game states each second. This is quite slow. For comparison, readily available Chess playing programs such as GNUchess and Crafty can easily search through over 35,000 game states each second and with the aid of specially designed Chess chips, Deep Blue is capable of examining over

200 million game states each second. Thankfully, the effectiveness of the evaluation function generated by OGRE compensates for this serious shortfall.

In practice, random play is sufficient to identify pieces in most games. However, this method can prove inadequate in some game definitions. It may be advantageous to pursue a learning strategy that includes active exploration. A system that purposefully explores new areas and expands unvisited branches of the game tree might perform better.

11. SUMMARY OF PART I

This part of the dissertation presented a novel method for automatic feature extraction and an implementation of this method in the form of OGRE, a successful General Game Playing system.

Chapters 3 through 6 covered introductory material. Chapter 3 introduced several concepts related to the topic of Computer Game Playing. Chapter 4 described General Game Playing (GGP) and outlined some of the challenges related to this area of research. Chapter 5 covered the topic of Feature Extraction which is a key component of General Game Playing while Chapter 6 introduced the topic of Theorem Proving.

Chapters 7 through 10 covered our implementation of a General Game Playing (GGP) system, OGRE. Chapter 7 presented the implementation architecture and design issues behind OGRE, a General Game Playing agent that participated in the AAAI-2006 General Game Playing competition. Chapter 8 described our innovative method for extracting important features such as pieces from the game definition. These features are important for a GGP agent to play effectively in games it has never seen before. Chapter 9 describes how we combine these features into a specialized evaluation functions for previously unfamiliar games. Chapter 10 presents the results of OGRE in the AAAI 2006 competition. The innovative method for feature extraction allowed our system to come in fourth place out of twelve participants.

Part II of this dissertation will focus on the structure of games and outline a framework that will allow for experimentation with a wide variety of games.

12. INTRODUCTION TO PART II

Part II introduces a framework for testing games and a language to describe them: the Regular Game Language (RGL). RGL was designed to address the fact that there is no standard formal method to describe either arbitrary stochastic games or arbitrary games of imperfect information.

Chapter 13 introduces several areas of research that are related to games and outlines structures that are important in the definition and analysis of games. Chapter 14 introduces the Regular Game Language. Chapter 15 describes the syntax of the Regular Game Language.

13. GAMES

Games have captivated people for thousands of years. Games and puzzles (one-player games) have been an inspiration in many areas of research ranging from Statistics to Graph theory. Games are used extensively in areas such as Game Theory, Game Logic, Game Semantics, and Computational Game Theory. However, these theories look at games from an abstract point of view.

13.1 Game Theories

Game Theory studies the ways in which strategic interactions among rational agents produce outcomes with respect to their preferences. Game Theory was founded in the early twentieth century by Zermelo, Borel, and von Neuman on parlor games. John Nash made his famous contributions to non-cooperative game theory in the 1950's. Game Theory is not concerned with the details of any particular contest, only the results and their utility to the individual participants (Williams 1986).

Game Logic focuses on the choices made by players, but not on the details of what the moves entail. Developed by Parikh (1985) for reasoning about neighborhood models, Game Logic extends propositional logic by adding modal operators whose meanings are assigned by games. Games in extensive form are modeled as trees whose nodes are possible states of the game, end nodes being terminal game states. Labeled directed edges from a node to its children indicate available moves. Game Logic has more detail about a game than Game Theory. It is concerned with the moves of each player as opposed to the strategy of the players. However the theory does not concern itself with the details of those moves.

Game Semantics interprets computation as a dialog between two parties. The idea is simple: a logic formula is interpreted as a game between two players the "Verifier" and the "Falsifier". The two players take turns attempting to refute or defend the validity of the formula. The formula is semantically valid if there is a strategy by which the "Verifier" can always win: a winning strategy (Abramsky and McCusker 1998). Game Sematics approaches games at the same level of detail as Game Logic.

Combinatorial Game Theory (Berlekamp, Conway and Guy 2001) studies the strategies and mathematics of a large range of perfect information two-player games that involve no chance elements. The theory relates the moves of each player to John H. Conway's surreal numbers (Conway 1976) and leads to several interesting complexity results.

Conway's Game of Life is a cellular automaton (Gardner 1970). The game is played on an infinite grid of squares cells, where each cell has one of two states, dead or alive. The rules are as follows:

1) If a cell is alive and two or more adjacent cells are alive, then the cell stays alive, otherwise the cell's state changes to dead.
2) If a cell is dead and exactly three surrounding cells are alive, then the cell's state changes to alive, otherwise it remains dead.

The game can be thought of as a zero-player game; it requires no player intervention after the initial state is determined. Or said another way, there are no options during the course of play; everything is completely deterministic. This simple game has been shown to have the power to emulate a universal Turing Machine (Rendal 2002). A Turing Machine is a construct that can perform any finite algorithm.

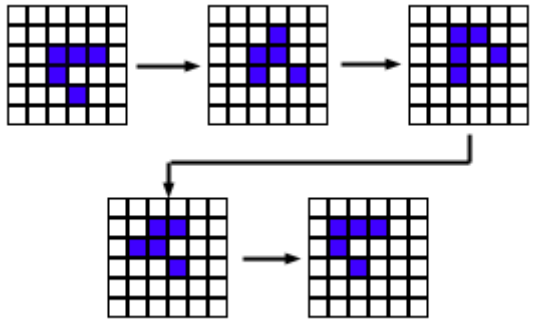

Figure 16. A "Glider" in Conway's Game of Life

Games seem to encapsulate something more than Turing Machines. Computationally they are more akin to Interaction Machines (Wegner 1997) or Computability Logic (Japaridze 2003, 2006). Interaction Machines are automata which are extensions of Turing machines in that they can interact with their environment with new input and output actions. Computability Logic is a mathematical framework for redeveloping logic as a formal theory of computability as opposed to classical logic which is a formal theory of truth. Computability Logic deals with computational problems as games played by a machine against the environment.

13.2 Games Taxonomy

Games can be classified from several perspectives and in many dimensions. For example, Burns (1998) groups games into the categories shown in Figure 17. A mathematical perspective focuses on information and probability. Game designers might concentrate on process. From a player's point of view, a taxonomy of games focuses on structure.

- **Card games**
  - **Patience Games**
  - **Gambling Games**
  - **Non-Trick Games**
  - **Trick Games**
  - **Children's Games**
- **Board Games**
  - **Family Board Games**
  - **Race Games**
  - **War Games**
  - **Territorial Games**
- **Domino & Dice Games**
  - **Domino Games**
  - **Dice Games**
- **Family Games**
  - **Parlor Games**
  - **Paper & Pencil Games**
  - **Word Games & Spoken Games**
  - **Written Games**
- **Sporting & Active Games**
  - **Games of Skill**
  - **Outdoor Games**

Figure 17. The game categories of Burns

13.3 High Level Abstraction

Systems that model games at a high level of abstraction, like Game Theory and the Theory of Computation use three common features to classify games: players, randomness, and information. The number of players, whether a game involves random chance and whether or not players have access to complete information about a particular game are the three topics we cover in this section.

Game Theory uses three common features to classify games: players, randomness, and information. The Theory of Computation includes analysis of games that involve one or two players, randomness and resource bounds. Examples include Condon's game

automata, games against nature, and Arthur-Merlin games (Condon 1989). Games between two players, of the kind where one player wins and one loses, are used in many branches of logic. Important examples are semantic games used to define truth, back-and-forth games used to compare structures, and dialogue games to express formal proofs.

13.4 Players

The number of players is fundamental to any game and a natural place to start classification. This is the approach taken in game theory as seen in Figure 18. Game Theory categorizes games into three categories: games of skill, games of chance, and games of strategy. Games of skill are one-player games where the single player has complete control over all the outcomes. Taking a test is one example. Solving a Towers of Hanoi puzzle is another. One-player games are often referred to as puzzles.

Games of chance are one-player games against nature. Unlike games of skill, the player does not control the outcomes completely and strategic selections do not lead unavoidably to certain outcomes. The outcomes of a game of chance depend partly on the player's choices and partly on nature, which is treated as a disinterested second player. In Game Theory games of strategy are games involving two or more players, not including nature, each of whom has partial control over the outcomes (Williams 1986).

We will separate games into three categories as well, but slightly differently. The three categories are: one-player games, two-player games, and multi-player games. Zero-player games, such as Conway's game of Life do exist, but we will not cover them in our treatment.

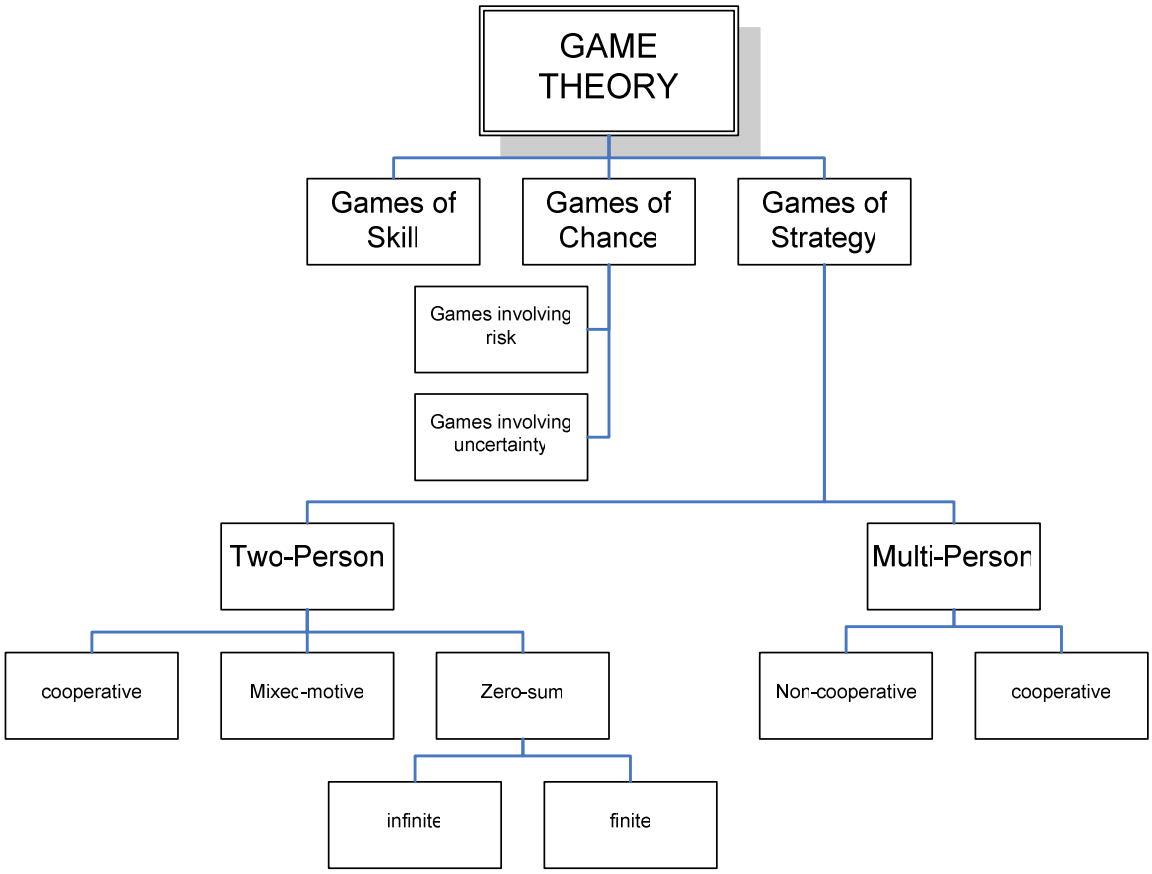

Figure 18. Game Theory.

13.5 Randomness

The element of chance is a key component of many games, from dice games like Backgammon to card games like solitaire. Game Theory distinguishes between "risk" and "uncertainty". The term "risk" refers to situations where the decision-maker can assign mathematical probabilities to the randomness which he is faced with. In contrast "uncertainty" refers to situations when the randomness cannot be expressed in terms of specific mathematical probabilities.

The randomness of games involving physical sports such as soccer and basketball are subject to uncertainty. Board games such as Backgammon and Poker involve risk. It is the calculable form of randomness to which we will focus our attention. In her

computational game model, Anne Condon (1989) refers to this as *Randomness.* Each player can make random choices, such as shuffling a deck of cards, rolling dice or flipping a coin.

13.6 Information

Games such as Chess or Checkers are known as *perfect information* games. Every participant is privy to all the information about the current game state. When players can keep some information private from the other players, as in games like bridge, the game is one of *imperfect information*. Sometimes these are known as partial information games.

13.7 Partisan vs. Impartial

Combinatorial game theory (Berlekamp, Conway and Guy 1982) deals abstractly with a very large range of two player games of perfect information with no randomness. A game is called *impartial* if both players have exactly the same moves (e.g. Nim). Games in which players have different move options are called *partisan*.

14. REGULAR GAME LANGUAGE

The Regular Game Language was designed to provide a framework for research in the area of games. It provides support for single and multiple players, imperfect and perfect information, as well as deterministic and stochastic games. The language also provides support for common game concepts such as pieces, locations, arithmetic functions and numbers.

14.1 Game Grammars

There have been a few methods proposed to define the rules of games in general. Barney Pell created a game definition grammar, Metagame (Pell 1995). Romien (Romein 2001) developed a language, Multigame, for expressing the rules of one and two-person board games. Orwant designed the EGGG language to codify game rules (Orwant 2000) and in (Kaiser 2000) we presented a new class of games called Simple War Games. More recently, the Game Description Language (Genesereth, Love and Pell 2005) has been used in the AAAI General Game Playing competition in both 2005 and 2006.

However, all these descriptive methods are limited. Metagame can only define games on a Chess-like grid; Multigame can only define two-person games; Simple War Games limit the types of piece movement. Although GDL has not been formally analyzed, the language appears to have the most expressive power. It is able to describe most perfect information, deterministic games of any number of players.

Unfortunately, none of these languages provide support for defining games of *imperfect information* and, with the exception of Simple War Games, do not support games with *random* elements. That means that games such as Backgammon, Poker,

Battleship and Scrabble are outside the domain of these previous frameworks. It would be advantageous to include these elements.

Perfect-information domains are the exception and not the rule in the real world. The richness of a domain that includes games of imperfect information and games with random elements make it superior to traditional environments for exploring many issues needed to achieve the ultimate goal of near-human-like Artificial Intelligence.

Games with imperfect information and randomness provide researchers with the test-bed to explore concepts such as deception, opponent modeling and information sparsity (Billings, Davidson, Schaeffer, and Szafron 2002). Clever opponents are able to exploit predictable play, so deception can play a key role in some games such as Poker and Diplomacy. Opponent modeling involves observing your opponents and adjusting play to exploit their perceived weaknesses. In card games the amount of information that a player has may be limited, even at the end of the game, which leads to learning challenges. In Poker, play may end with players not having to reveal their cards. This information sparsity limits the amount of data from which to learn.

To provide a framework for exploration in these interesting areas, we developed the Regular Game Language.

14.2 RGL

The Regular Game Language was designed to provide a framework for research in the area of games. It provides support for single and multiple players, imperfect and perfect information, as well as deterministic and stochastic games. The language also provides support for common game concepts such as pieces, locations, arithmetic functions and numbers.

Informally a game consists of players, something for the players to manipulate, which we will refer to as pieces, a place for the pieces to be located, which we will call a board, and finally a set of rules for manipulating the pieces. Generally speaking the rules can be thought of as algorithms or a set of instructions that all players must follow to play the game. For each game there are a set of rules outlining what each player can do at any particular time in the game, when the game is over and who wins. Players play the game by following these instructions.

In our model, the entire state of the game is represented in the form of pieces and their locations. Players change the state of the game by moving the pieces around on the board.

14.3 The Pieces

Most board games have small objects that the players place on or move around the board. In our model these are called pieces, whether they are rooks, checkers, disks, tokens, stones, Xs or Os.

In some games the pieces themselves indicate state information for the game by how they are placed on the board. Othello pieces have two sides and the side facing up indicates the current state of that piece. For our purposes, state information will not be stored in this manner. For example rather than flipping pieces over in an Othello game, a black piece would be replaced by a white piece or visa versa.

In our model, pieces do have static attributes. To support games that include imperfect information, these attributes can be either hidden, or made visible to specific or all players. For example in a card game, a piece might have a ‘face’ and a ‘back’ where the ‘back’ is visible to all players, while the ‘face’ is visible to only one.

Pieces need to be located somewhere. In our model, each piece is located at a single location. In games like Monopoly or Risk, more than one piece may be located at a single location, while games such as Go or Hex restrict each location to a single piece. Our model supports both of these possibilities.

14.4 The Board

The playing surface, or board, of a game is represented by a labeled directed graph. Both the nodes and the edges are labeled. The nodes are the locations where pieces are placed. The game pieces are represented as distinguished symbols at the nodes.

Many popular games are played on a grid. Chess and Tic-Tac-Toe are both played on a grid consisting of a lattice of squares. In games like Hex or Chinese Checkers the board is a lattice of hex shaped locations instead of squares. Board games like Risk or Diplomacy have locations that are neither uniform in size nor regular or consistent in the way they are connected to one another. But square grid boards, hex grid boards and unstructured boards can all be easily represented by directed labeled graphs.

In games such as Risk where the direction of the connection is unimportant, the labels can be omitted without adverse effect. However, labels are helpful in games like Monopoly or Draughts to avoid an explosive increase in the number of rules necessary to describe valid moves.

The nodes are the locations where pieces are placed. By labeling the edges, we are able to express more complex relationships than simple adjacency. As a simple example, consider the graph for Tic-Tac-Toe in Figure 19. The nodes are represented as circles {1 … 9} and edges are labeled with lowercase letters {h, v, d, a}.

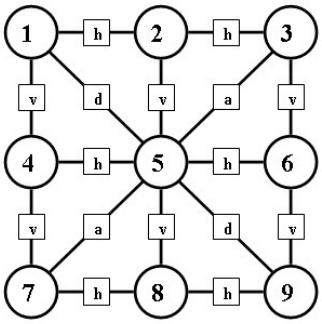

Figure 19. The graph for Tic-Tac-Toe.

Each of the edge labels encapsulates the notion of a type of relationship between board locations. A string of locations connected by edges with the same label are in a 'line'. For example, the notion of horizontal rows is encapsulated in the edges labeled 'h'.

15. REGULAR GAME LANGUAGE SYNTAX

This section provides a detailed description of the syntax and semantics of the Regular Game Language. RGL is based on the familiar syntax of Prolog. To be precise, RGL is Prolog with a distinguished set of statements. Prolog uses a single data type called a *term*. A *term* can be:

- Constant symbol
  - Atoms (names)
  - Numbers (integers)
- Variable symbol
- The anonymous variable symbol "_" (the underscore)
- Compound term

Logical variables start with an upper-case letter, atoms with a lower-case letter, and numbers with a digit. Logical variables are untyped, and can be instantiated by substitution by another term. Instantiation occurs by unification through pattern matching and appearing in an equality expression. The anonymous variable (the underscore) matches or unifies with anything and is used when it does not matter what the variable matches. A compound term has a *functor* and a number of arguments which are also terms. The number of arguments is called the term's *arity*. Examples for compound terms are foo(a,b), bar(x,y,z) or foo(foo(a,b),a).

Prolog programs are defined by *clauses*. A clause has two components, the head and the body. Clauses with empty bodies are called facts. An example of a fact is:

```
player(red).
```

If a clause's body is not empty it is called a rule. An example of a rule is:

```
score(x,100):- threeinarow(x).
```

The ":-" means "if". This rule means that player "x" scores 100 if the fact "threeinarow(x)" is true. Prolog includes the *Unique Names Assumption* which says that any two terms with different names are, in fact, distinct entities (e.g. "Miami" and "Hollywood" are assumed to be different). Prolog also includes the *Domain Closure Assumption* which says that the only objects in the universe are those named by ground terms.

RGL has the following set of distinguished relations: `player, node, edge, initVisible, pieceAt, score, gameover`. This section provides a more detailed description of the Regular Game Language statements. We will use the familiar game of Tic-Tac-Toe to illustrate the Regular Game Language constructs. Also referred to as Noughts and Crosses, Tic-Tac-Toe is a game for two players X and O, who take turns to mark spaces in a 3x3 grid. The player who makes three of their marks in a row, horizontally, vertically or diagonally, wins the game.

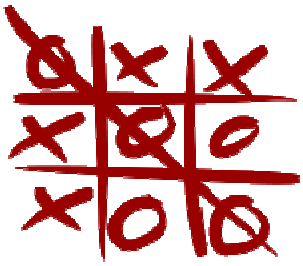

Figure 20. A game of Tic-Tac-Toe where player O has won.

15.1 The Players

The RGL defines the players of the game through the *"player"* statement. In Tic-Tac-Toe, the statement is:

```
player(playerx).
player(playero).
```

This indicates that the game has two players, referred to in the game as `playerx` and `playero`.

15.2 Board

The game board is represented by a graph. The two statements *"node"* and *"edge"* are used to define graphs. The graph for Tic-Tac-Toe is define as follows

```
node(a/1).
node(a/2).
node(a/3).
node(b/1).
node(b/2).
node(b/3).
node(c/1).
node(c/2).
node(c/3).
node(whoseturn).

edge(a/1,a/2,h).
edge(b/1,b/2,h).
edge(c/1,c/2,h).
edge(a/2,a/3,h).
edge(b/2,b/3,h).
edge(c/2,c/3,h).
edge(a/1,b/1,v).
edge(a/2,b/2,v).
edge(a/3,b/3,v).
edge(b/1,c/1,v).
edge(b/2,c/2,v).
edge(b/3,c/3,v).
edge(a/1,b/2,d).
edge(b/2,c/3,d).
edge(a/3,b/2,a).
edge(b/2,c/1,a).
```

Note that there is also a node "`whoseturn`" in this definition which we will use to identify whose turn it is to play. There is a shorthand method for defining nodes and edges.

```
nodes([a/1, a/2, a/3, b/1, b/2, b/3]).
nodes([c/1, c/2, c/3, whoseturn]).
edges([[a/1,a/2,h],[b/1,b/2,h],[c/1,c/2,h]]).
edges([[a/2,a/3,h],[b/2,b/3,h],[c/2,c/3,h]]).
edges([[a/1,b/1,v],[a/2,b/2,v],[a/3,b/3,v]]).
edges([[b/1,c/1,v],[b/2,c/2,v],[b/3,c/3,v]]).
edges([[a/1,b/2,d],[b/2,c/3,d]]).
edges([[a/3,b/2,a],[b/2,c/1,a]]).
```

Because many popular board games are played on grid lattices shorthand statement *"grid"* is included in RGL. This statement significantly reduces the number of statements needed to define the game board. The statement `grid(3,3)` implicitly defines nodes `a/1 through c/3`.

```
grid(3,3).
nodes([whoseturn]).
```

15.3 Pieces

Pieces and their relationship to one another on the game board represent the current state of the game. Players move these pieces around during the game. In certain games, particularly those with hidden information such as card games, some attributes of the pieces (for example the suit and rank) may be hidden from the opposing player. The example game Tic-Tac-Toe contains only two pieces with no hidden information.

```
attrImage(x,front,'x').
attrImage(o,front,'o').
attrImage(x,back,'x').
attrImage(o,back,'o').
```

15.4 Visibility

Visibility in RGL is modeled by giving pieces attributes, and then controlling access to this information. The "*initVisible*" statement defines the initial visibility all the pieces in a particular node on the game board.

```
initVisible(whoseturn,front,all).
initVisible(whoseturn,back,all).
```

Our example game is one of perfect information and therefore all players are privy to all information about each of the pieces on the board. In this case the visibility of the piece that controls the active player is made visible to all players.

15.5 Initial State

The *"init"* statement indicates the initial state of the game. Some games, such as Checkers and Chess, have many pieces that start on the board. Other games, like Go or Tic-Tac-Toe, have an empty initial game board. Recall that in our model, the entire game state is represented on the board. So information such as the player to move next, must also be indicated on the board. Therefore our example game of Tic-Tac-Toe has the following init statement:

```
init(whoseturn,[x]).
```

This indicates that a piece labeled "`x`" is located at the node labeled "`whoseturn`". It will be used to indicate which player is currently moving.

15.6 Randomness

The *"shuffle"* and *"roll"* statements are used to provide support for random information. The *"shuffle"* statement takes all the pieces at a particular node on the game board and shuffles them around so that they are not in the same order. The *"roll"* statement places a random piece at the indicate position on the board. Tic-Tac-Toe does not involve random information.

15.7 Query functions

RGL provides several predefined functions to query the state of the game. These functions make defining a game more manageable by reducing the number of functions that game designers need to create.

`pieceAt(P,N)` – if both `P` and `N` are bound variables then the statement resolves to true if piece `P` is at node `N` of the game board and false otherwise. If either variable is unbound, the system returns with the appropriate variable bindings or with false if no such bindings can be found.

`onboard(N)` – resolves to true if `N` is a node on the game board, and false otherwise. If either variable is unbound then this function can be used to return all positions on the game board.

15.8 Movement

Movement is the core of any game. Players change the state of the game by moving pieces around on the board. At its core, the RGL allows only two fundamental actions for manipulating the game state: adding and removing. A piece can either be added to the game board, or removed from the game board. These two functions are sufficient in and of themselves for many games with very simple structures like Tic-Tac-Toe, Nim, or Hackenbush. But more complex actions can be formed by combining these simple functions.

Conceptually a movement operation is a quadruple <*Y,A,P,R*> in which each component has the following meaning:

1. *Y* is the player making the move.
2. *A* is the action identifier.

3. *P* is a set of preconditions that must be met before the move can be made. The preconditions in set *P* can be any conditional statement in the language.
4. *R* is the set of changes that will be made to the game state.

This method uses many representational assumptions consistent with the STRIPS language (Fikes and Nilsson 1971). The STRIPS (Stanford Research Institute Problem Solver) language is the base for most languages for expressing automated planning problems instances. Structurally movement operators in the game language we will be discussing shortly have some similarity to actions in the STRIPS planning language.

```
Pickup (X)
Preconditions: Clear(X)
               OnTable(X)
               HandEmpty
Add list:      Holds(X)
Delete list:   HandEmpty
               Clear(X)
               OnTable(X)
```

Figure 21. Example STRIPS action.

One notable difference is that STRIPS makes the simplifying assumption that actions can always be executed but they do not have effects if their preconditions are not met. This is not the case in RGL where all preconditions must be met before a player can make the move.

Syntactically, the moves are defined in a pair of statements: *"moveprecon"* and *"moveresult"*. The *"moveprecon"* statement indicates what conditions must be met before the move can be made. The *"moveresult"* statement lists what the results will be. Even a simple movement may cause multiple effects. For example, jumping an opposing piece in Checkers will result in the movement of the piece, the removal of the opposing

piece, and possibly promoting the moving piece to a king. The format of *"moveprecon"* and *"moveresult"* is shown here:

```
moveprecon(<player>,<action>):-
     <preconditionlist>.
moveresult(<player>,<action>,<resultlist>):-
     <conditionlist>.
```

Where the `<player>` term is either a variable, or one of the terms defined in a `player(<player>)` relation. The `<action>` term is a relation of the form

```
functor(<Piece>,<Location>)
```

or

```
functor(<Piece>,<Location1>,<Location2>)
```

while `<preconditionlist>` and `<conditionlist>` are a lists of valid relations.

Note that the terms `<player>` and `<action>` must match in both of the two statements *"moveprecon"* and *"moveresult"*, otherwise there will be no results for an action and/or no legal precondition for the mismatched result.

The *movement and visibility operators* in RGL are the only relations that can be in the *<resultList>* of the *"moveresult"* statement. The *movement operators* are: `place`, `remove`, `move`, `replace`. The movement operators are the functions that actually change the state of the game by moving pieces.

`place(P,T)` – places a piece `P` at the given location `T`.

`remove(P,F)` – removes the specified piece `P` from the location `F`.

`move(P,F,T)` – moves the specified piece `P` from one location `F` to another location T. This function is just a shorthand way of writing `remove(P,F)` and `place(P,T)`.

`replace(P,T,N)` – removes piece `P` from location `T`, and replaces it with piece `N`. This function is just a shorthand way of writing `remove(P,T)` and `place(N,T)`.

Another relation that is restricted to the `<resultList>` of the *"moveresult"* statement is the *"reveal"* statement. To provide support for games with hidden information, all attributes of all pieces are considered hidden from all players until these attributes are revealed to them. This *visibility operator* allows players to become aware of the attributes of specific pieces. It does not actually change the state of the game, but it can change a player's perception of the game state.

`reveal(Y,P,T,A)` – reveals to player Y, the attribute A of piece P at location T.

The movement and visibility operators are the only relations allowed in the `<resultList>` of the *"moveresult"* statement and these operators cannot appear in any other context except the `<resultList>` of the *"moveresult"* statement.

Movement for the example game of Tic-Tac-Toe is shown in Figure 22, Figure 23, and Figure 24.

```
moveprecon(Player,place(Piece,To)):-
     whoseturn(Player)
     owner(Player,Piece)
     onboard(To)
     not(pieceAt(_,To)).
```

Figure 22. Precondition rules for placing a mark.

The active player is allowed to place their own piece on the board in any location that does not already contain a piece. Notice in Figure 22 that the `onboard(To)` and `pieceAt(_,To)` conditions are both distinguished conditional relations in RGL. The `whoseturn(Player)` and `owner(Player,Piece)` conditions are specific to this game definition.

```
moveresult(Player,place(Piece,To),
    [
        place(Piece,To),
         reveal(all,Piece,To,front),
         replace(Player,whoseturn,Next),
         reveal(all,Next,whoseturn,front)
        ]
       ):-
    nextPlayer(Player,Next).
```

Figure 23. Result rules for placing a mark in Tic-Tac-Toe.

The result of placing a piece (as described in Figure 23) is to a) place the piece, b) reveal this fact to all the players, c) replace the piece indicating whose turn it is, d) reveal whose turn it is to all players.

```
% ----------------
% -- PASS
moveprecon(Player,pass):-
    not(whoseturn(Player)).

moveresult(Player,pass,[]).
```

Figure 24. Rules for non-active player to pass.

In our example game, players do not move simultaneously; only one player is allowed to move at one time. Therefore the statements given in Figure 24 allow the inactive player to pass.

15.9 Game Over

The *"gameover"* statement indicates the termination conditions under which the game ends. In Tic-Tac-Toe, the game terminates when a player has made three marks in a row or when the board is full and players have no more legal moves.

```
gameover:- threeinarow(_).
gameover:- not(legal(_,_)).
```

The *"score"* statement indicates how each player should view the results of a game. Each player is assigned an integer value from between 0 and 100. Zero being bad and 100 being defined as good. For games where one player wins and another loses, the "winning" player could receive 100 points while the other player receives zero points. The 100 points can also be divided amongst the players in other ways. Cooperative games might assign identical scores to players on the same "team". Single player puzzles might assign credit for partially completing goals. Multi-player games can assign progressive amounts to 1st, 2nd and 3rd place.

Continuing our Tic-Tac-Toe example, the following statements indicate the score for each player. Tic-Tac-Toe games often end in ties, and in this definition each player is awarded 50 points when that occurs.

```
score(playerx,100):- threeinarow(x).
score(playerx,0)  :- threeinarow(o).
score(playerx,50).
score(playero,100):- threeinarow(o).
score(playero,0)  :- threeinarow(x).
score(playero,50).
```

Note that the function "`threeinarow(_)`" is not part of the language definition but rather a helper function defined by the game designer specifically for this implementation of the game.

16. SUMMARY OF PART II

This part of the dissertation has discussed the structure of games and the general issues involved in construction of a language to support all board games. Chapter 13 introduced several areas of research that are related to games and outlines structures that are important in the definition and analysis of games. Chapter 14 introduced the Regular Game Language which is capable of describing not only two player, perfect information games such as Chess and Checkers, but also stochastic games such as Backgammon, and imperfect information games such as Bridge and Stratego. Chapter 15 described the syntax of the Regular Game Language. The Regular Game Language will provide a useful framework for future research in the area of General Game Playing.

17. CONCLUSIONS

Artificial Intelligence research in Computer Game Playing continues to expand beyond the narrow confines of classic board games such as Chess and Checkers. The Game Description Language and the recent AAAI General Game Playing and AAAI Computer Poker competitions epitomize this interest.

This dissertation has two main contributions. It first describes the design and implementation of a successful General Game Playing system. The main contribution here is an original technique for automatically identifying critical game features by examination of the game description and through self play. This method has been incorporated into a fully autonomous agent that participated successfully in the second AAAI General Game Playing competition which was held at the AAAI 2006 in Boston. Our system, OGRE, came in fourth place.

Secondly we introduce the Regular Game Language which is the first General Game Playing language to provide support for single and multiple players, imperfect and perfect information, as well as deterministic and stochastic games. The Regular Game Language will provide a useful framework for future research in the area of General Game Playing. This framework can be used as the basis for machine learning in games to provide comparisons between the games and to explore the effects of variations on learning techniques.

REFERENCES

Abramsky, S. & McCusker, G. (1998). Game Semantics. In H. Schwichtenberg & U. Berger (Eds.) *Logic and Compuation: Proceedings of the 1997 Marktoberdorf Summer School*, Springer-Verlag.

Allis, L. V. (1994). *Searching for solutions in games and artificial intelligence.* (Doctoral dissertation, University of Limburg, Maastricht, The Netherlands).

Berlekamp, E. R., Conway, J. H., & Guy, R. K. (2001) *Winning Ways for your Mathematical Plays* (2nd ed.). Natick, MA: A.K. Peters.

Billings, D., Davidson, A. Schaeffer, J. Szafron, D. (2002) The Challenge of Poker. *Artificial Intelligence* 134(1-2): 201-240.

Bratko, I. (1990). *Prolog programming for artificial intelligence* (2nd ed.). Wokingham, England: Addison-Wesley.

Burns, B. (1998) (ed.) *The Encyclopedia of Games*. London: Brown Packaging Books, Ltd.

Campbell, M., Hoane, A. J., Jr., & Hsu, F. (2002, January). Deep Blue. *Artificial Intelligence*, 134(1-2), 57-83.

Condon, A. (1989). *Computational models of games*. Cambridge, MA: MIT Press.

Conway, J. H. (1976). *On Numbers and Games*. London: Academic Press.

Epstein, S. (1994). Identifying the right reasons: Learning to filter decision makers. In *Proceedings of the AAAI 1994 Fall Symposium on Relevance*, 68-71. New Orleans, LA: AAAI Press.

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

Fikes, R. & Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proceedings of Second International Joint Conference on Artificial Intelligence* (pp. 608-620). Los Altos, CA: William Kaufmann, Inc.

Fikes, R., Frank, G., & Jenkins, J. (2003). JTP: A system architecture and component library for hybrid reasoning. In *Proceedings of the 7th World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando, Florida, USA.

Fürnkranz, J. (2001). Machine learning in games: A survey. In J. Fürnkranz & M. Kubat (Eds.) *Machines That Learn to Play Games* (pp. 11-59). Commack, NY: Nova Science Publishers.

Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223(4), 120-123.

Genesereth, M. (1991). Knowledge Interchange Format. In: J. Allen, R. Fikes, & E. Sandewall (Eds.) *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference* (KR'91). San Fransisco, California: Morgan Kaufmann Publishers.

Genesereth, M., Love, N., & Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI Magazine,* 26(2), 62-72.

Gherrity, M. (1993) *A Game-Learning Machine*. (Doctoral dissertation, University of California, San Diego, California).

Gould, J. & Levinson, R. (1992) Experience-Based Adaptive Search, In R. S. Michalski & G. Tecuci (Eds.) *Machine Learning: A Multi-Strategy Approach*, (Vol 4.), pp. 579-604. San Fransisco, CA: Morgan Kaufman

Greenblatt, R., Eastlake, D., & Crocker, S. (1967). The Greenblatt Chess Program, *Proceedings of the AFIPS Fall Joint Computer Conference* (31), 801-810. Reprinted (1988) in Computer Chess Compendium (ed. D.N.L. Levy), pp. 56-66. New York: Springer-Verlag Inc.

Guyon, I., & Elisseeff, A. (2006). An Introduction to Feature Extraction. In: I. Guyon, S. Gunn, M. Nikravesh, & L. Zadeh (Eds.), *Feature Extraction, Foundations, and Applications*. Series Studies in Fuzziness and Soft Computing. New York: Springer.

Halck, O. M. & Dahl, F.A. (1999). On Classification of Games and Evaluation of Players – With Some Sweeping Generalizations About the Literature. In: J. Fürnkranz & M. Kubat (Eds.), *Proceedings of the 16th International Conference on Machine Learning (ICML-99) Workshop on Machine Learning in Game Playing*, Ljubljana, Slovania: Jozef Stefan Institute.

Japaridze, G. (2003). Introduction to Computability Logic. *Annals of Pure and Applied Logic* 123, (pp. 1-99).

Japaridze, G. (2006). Computability Logic: A Formal Theory of Interaction. In D. Goldin, S. Smolka & P. Wegner (Eds.) *Ineractive Computation: The New Paradigm*. (pp 183-223) Berlin: Springer-Verlag.

Kaiser, D. (2000). A Generic Game Playing Machine, unpublished master's thesis, Florida International University, Miami, Florida.

Kaiser, D. (2005). The Structure of Games, In *Proceedings of the 43rd Annual Southeast Regional Conference*. (Kennesaw, Georgia, March 18 – 20, 2005) ACM-SE 43. ACM Press.

Kaiser, D. (2007a). The Design and Implementation of a Successful Autonomous Game Playing Agent. *Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference*. FLAIRS 2007. May 7-9, 2007 Key West, Florida, USA

Kaiser, D. (2007b). Automatic Feature Extraction for Autonomous General Game Playing Agents. *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS'07. May 14-18, 2007 Honolulu, Hawaii, USA

Knuth, D. E. (1973). *The Art of Computer Programming. Volume 3: Sorting and Seaqrching*. Reading, MA: Addison-Wesley Publishing Company.

Knuth, D. E. (1988). *The Art of Computer Programming*, *volume 2: Seminumerical Algorithms*, (3rd ed.), p 232. Boston: AddisonWesley.

Korf, R., Reid, M., Edelkamp, S. (2001, June). Time Complexity of Iterative Deepening-A*, *Artificial Intelligence*, 129(1-2), 199-218.

Kuhlmann, G., Dresner, K., & Stone, P. (2006). Automatic Heuristic Construction in a Complete General Game Player. In *Proceedings of the Twenty First National Conference on Artificial Intelligence*. 1457-1462.

Levinson, R. (1994). *MORPH II: A universal agent: Progress report and proposal*. Technical Report Number UCSC-CRL-94-22, Department of Computer Science, University of California, Santa Cruz, Jack Baskin School of Engineering.

Levinson, R. (1995). *General game playing and reinforcement learning*. Technical Report, Number UCSC-CRL-95-06, Department of Computer Science, University of California, Santa Cruz, Jack Baskin School of Engineering..

Levy, D. N. L., & Newborn, M. M. (1991). *How Computers Play Chess*. New York: W.H. Freeman and Company.

Love, N. (2006). General Game Playing Competition Results. Retrieved September 24, 2006, from http://games.stanford.edu/2006results.html

Luckhardt, C. & Irani, K. (1986) An Algorithmic Solution of n-Person Games. In Proceedings of the Ninth National Confernce on Artificial Intelligence (AAAI-86): 158-162.

Mallet, J. & Lefler, M. (2007) Zillions of Games. Retreived on June 24, 2007 from http://www.zillions-of-games.com/

Orwant, J. (2000). EGGG: Automated programming for game generation. *IBM Systems Journal*, 39(3/4): 782-794.

Parikh, R. (1985). The Logic of Programs: New Directions. In M. Karpinski and J. van Leeuwen (Eds.) *Topics in the Theory of Compuations, Annals of Discrete Mathematics* 24, Elsevier.

Pell, B. (1993). Strategy Generation and Evaluation for Meta-Game Playing. (Doctoral dissertation, University of Cambridge, Cambridge, UK).

Pell, B. (1995). A Strategic Metagame Player for General Chess-Like Games. *Computational Intelligence, 11(4)*, 1995.

Rendal, P. (2002). Turing universality of the game of life. In A. Adamatzky (ed.) *Collision-Based Computing*. (pp. 513-539) London: Springer-Verlag.

Riazanov, A. & Voronkov, A. (2003). Limited Resouce Stategy in Resolution Theorem Proving, *Journal of Symbolic Computation*, 36(1-2).

Romein, J. (2001). *Multigame - An environment for Distributed Game-Tree Search*. (Doctoral dissertation, Faculty of Sciences, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdame, the Netherlands).

Russel, S., & Norvig, P. (2003). *Artificial Intelligence, A Modern Approach* (2nd Ed.), Prentice Hall International, Inc.

Sakuta, M., & Iida, H. (2000). Solving Kriegspiel-Like Problems: Exploiting a Transposition Table, *ICGA JOURNAL*, 23(4): 218-229.

Samuel, A. (1959, March 03). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development* (3), 210-229.

Schaeffer, J., Treloar, N., Lu, P., & Bryant, M. (1994). Chinook: The Man-Machine World Checkers Champion. *AI Magazine.* 17(1): 21-29.

Schulz, S. (2001). System Abstract: E 0.61. *Proceedings of First International Joint Conference on Automated Reasoning* (JCAR), Siena Italy, (June 18-23) 370-375.

Shannon, C. (1950, March). Programming a Computer for Playing Chess. *Philosophical Magazine*, Ser. 7, 41(314).

Sturtevant, N. R., & Korf, R. E. (2000). On pruning techniques for multi-player games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence* (July 30 – August 03, 2000), 201-207. AAAI Press / The MIT Press.

Sutcliffe, G. & Suttner, C. (1998). The TPTP Problem Library: CNF Release v1.2.1, *Journal of Automated Reasoning*, 21(2), 177-203.

Sutcliffe, G., Fuchs, M. & Suttner, C. (2000). *Progress in Automated Theorem Proving, 1997-1999*. Technical Report TR-ARP-11-00, Australian National University.

Sutcliffe, G. (2001). The CADE-17 ATP System Competition, *Journal of Automated Reasoning*, 27(3), 227-250.

Tammel, T. (1997). Gandalf. *Journal of Autmated Reasoning*. 18(2), 199-204.

Utgoff, P.E. (2001). Feature construction for game playing. In J. Fürnkranz & M. Kubat (Eds.), *Machines That Learn to Play Games*. 131-152. Commack, NY: Nova Science Publishers.

Voronkov, A. (1994) Implementing Bottom-up Procedures with Code-Trees: A Case Study of Forward Subsumption. Technical Reprot no. 88. Computing Science Department, Uppsala University.

Wegner, P. (1997). Why Interaction is More Powerful than Algorithms. *Communications of the ACM* 40(5).

Welford, B. P. (1962). Note on a method for calculating sums of squares and products. *Technometrics* 4(3): 419-420.

Williams, J. D. (1986). *The Compleat Strategyst: Being a primer on the Theory of Games of Strategy*. New York: Dover Publications Inc.

Zobrist, A. L. (1970). A New Hashing Method with Application for Game Playing. Technical Report #88, Computer Science Department, The University of Wisconsin, Madison WI, USA Reprinted (1990) in *ICCA Journal*, 13(2), pp 69-73.

APPENDIX A – Sample Game Definitions

This section contains two sample games defined in the Regular Game Language: Tic-Tac-Toe, and Stratego.

```
/*
 *    The Game of TicTacToe
 */

/*****************************************************************
 ***                 DEFINITION
 *****************************************************************/

% ------------------------------
% define the players
player(x).
player(o).

% ------------------------------
grid(3,3).

otherlocations([whoseturn]).

whoseturn(P):- pieceAt(P,whoseturn).
nextPlayer(x,o).
nextPlayer(o,x).

%-----------------------------
% when is the game over.
gameover:- threeinarow(_).
gameover:- not(precon(X,Y)).

%-----------------------------
% who wins?
score(x,100):- threeinarow(x).
score(x,0)  :- threeinarow(o).
score(x,50).
score(o,100):- threeinarow(o).
score(o,0)  :- threeinarow(x).
score(o,50).

% -- init(+Location,+PieceList).
% -- shorthand version of piece creation.
init(whoseturn,[x]).

% -- initVisible(+location,+attr,+who).
initVisible(whoseturn,front,all).
initVisible(whoseturn,back,all).
```

```
/*****************************************************************
 ***                HELPER PREDICATES
 *****************************************************************/

threeinarow(P):-
      pieceAt(P,X/1),
      pieceAt(P,X/2),
      pieceAt(P,X/3).

threeinarow(P):-
      pieceAt(P,a/X),
      pieceAt(P,b/X),
      pieceAt(P,c/X).

threeinarow(P):-
      pieceAt(P,a/1),
      pieceAt(P,b/2),
      pieceAt(P,c/3).

threeinarow(P):-
      pieceAt(P,a/3),
      pieceAt(P,b/2),
      pieceAt(P,c/1).


/*****************************************************************
 ***                PIECES
 *****************************************************************/

%---------------------------------
% what do the pieces look like?
attrImage(x,front,'x').
attrImage(o,front,'o').
attrImage(x,back,'x').
attrImage(o,back,'o').

%---------------------------------
% define piece attributes
owner(x,x).
owner(o,o).


/*****************************************************************
 ***                MOVEMENT
 *****************************************************************/
% ---------------------------------------
% define placement
precon(Player,place(Piece,To)):-
      whoseturn(Player),
      owner(Player,Piece),
      onboard(To),
      not(pieceAt(_,To)).
result(Player,place(Piece,To),
      [
       place(Piece,To),
```

```
        reveal(all,Piece,To,front),
        reveal(all,Piece,To,back),
        replace(Player,whoseturn,Next),
        reveal(all,Next,whoseturn)
         ]
        ):-
       nextPlayer(Player,Next).

% ----------------
% -- PASS
precon(Player,pass):-
      not(whoseturn(Player)).
result(Player,pass,[]).

/********************** EOF ************************/
```

```
/*
 *    A small version of the Game of Stratego
 */

/*****************************************************************
 ***                 STRATEGO: DEFINITION
 *****************************************************************/

% ------------------------------
% define the players
player(r).
player(b).

% --------------------------------
% grid(X,Y)
grid(4,10).

otherlocations([redhand,bluehand,whoseturn]).

zone(redzone,[a/1,a/2,
              b/1,b/2,
              c/1,c/2,
              d/1,d/2
             ]).

zone(bluezone,[a/9,a/10,
               b/9,b/10,
               c/9,c/10,
               d/9,d/10
               ]).

inzone(Location,Zone):-
      zone(Zone,ZList),
      member(Location,ZList).

whosezone(r,redzone).
whosezone(b,bluezone).

whosehand(r,redhand).
whosehand(b,bluehand).

whoseturn(P):- pieceAt(P,whoseturn).
nextPlayer(r,b).
nextPlayer(b,r).

%----------------------------
% put some pieces on the board.

gameover:- not(pieceAt(bf,_)).
gameover:- not(pieceAt(rf,_)).
gameover:- whoseturn(r),not(precon(r,_)).
gameover:- whoseturn(b),not(precon(b,_)).

score(r,100):- not(pieceAt(bf,_)).
score(r,0)  :- not(pieceAt(rf,_)).
```

```
score(r,50).
score(b,100):- not(pieceAt(rf,_)).
score(b,0)  :- not(pieceAt(bf,_)).
score(b,50).

% -- init(+Location,+PieceList).
% -- shorthand version of piece creation.
%%init(redhand, [r1,r2,r3,r3,r4,r4,r4,
                  r5,r5,r5,r6,r6,r6,
                  r7,r7,r7,r8,r8,r8,
                  r9,r9,r9,rs,
                  rm,rm,rm,rm,rm,rm,rf]).
%%init(bluehand,[b1,b2,b3,b3,b4,b4,b4,
                  b5,b5,b5,b6,b6,b6,
                  b7,b7,b7,b8,b8,b8,
                  b9,b9,b9,bs,
                  bm,bm,bm,bm,bm,bm,bf]).
init(redhand, [r1,r2,r3,r4,r8,rs,rm,rf]).
init(bluehand,[b1,b2,b3,b4,b8,bs,bm,bf]).
init(whoseturn,[r]).

% -- initVisible(+location,+attr,+who).
initVisible(redhand,front,red).
initVisible(redhand,back,all).
initVisible(bluehand,front,blue).
initVisible(bluehand,back,all).
initVisible(whoseturn,front,all).
initVisible(whoseturn,back,all).


/*****************************************************************
 ***                STRATEGO: PIECES
 *****************************************************************/

%---------------------------------
% what do the pieces look like?
attrImage(b1,front,'1').
attrImage(b2,front,'2').
attrImage(b3,front,'3').
attrImage(b4,front,'4').
attrImage(b5,front,'5').
attrImage(b6,front,'6').
attrImage(b7,front,'7').
attrImage(b8,front,'8').
attrImage(b9,front,'9').
attrImage(bs,front,'S').
attrImage(bm,front,'M').
attrImage(bf,front,'F').

attrImage(r1,front,'1').
attrImage(r2,front,'2').
attrImage(r3,front,'3').
attrImage(r4,front,'4').
attrImage(r5,front,'5').
attrImage(r6,front,'6').
```

```
attrImage(r7,front,'7').
attrImage(r8,front,'8').
attrImage(r9,front,'9').
attrImage(rs,front,'S').
attrImage(rm,front,'M').
attrImage(rf,front,'F').

attrImage(b1,back,'B').
attrImage(b2,back,'B').
attrImage(b3,back,'B').
attrImage(b4,back,'B').
attrImage(b5,back,'B').
attrImage(b6,back,'B').
attrImage(b7,back,'B').
attrImage(b8,back,'B').
attrImage(b9,back,'B').
attrImage(bs,back,'B').
attrImage(bm,back,'B').
attrImage(bf,back,'B').

attrImage(r1,back,'r').
attrImage(r2,back,'r').
attrImage(r3,back,'r').
attrImage(r4,back,'r').
attrImage(r5,back,'r').
attrImage(r6,back,'r').
attrImage(r7,back,'r').
attrImage(r8,back,'r').
attrImage(r9,back,'r').
attrImage(rs,back,'r').
attrImage(rm,back,'r').
attrImage(rf,back,'r').

attrImage(b,front,'b').
attrImage(b,back,'b').
attrImage(b,front,'r').
attrImage(b,back,'r').

%----------------------------------
% define piece attributes
owner(b,b1).
owner(b,b2).
owner(b,b3).
owner(b,b4).
owner(b,b5).
owner(b,b6).
owner(b,b7).
owner(b,b8).
owner(b,b9).
owner(b,bs).
owner(b,bm).
owner(b,bf).
owner(r,r1).
owner(r,r2).
owner(r,r3).
```

```
owner(r,r4).
owner(r,r5).
owner(r,r6).
owner(r,r7).
owner(r,r8).
owner(r,r9).
owner(r,rs).
owner(r,rm).
owner(r,rf).

value(b1,1).
value(b2,2).
value(b3,3).
value(b4,4).
value(b5,5).
value(b6,6).
value(b7,7).
value(b8,8).
value(b9,9).
value(bs,s).
value(bm,m).
value(bf,f).
value(r1,1).
value(r2,2).
value(r3,3).
value(r4,4).
value(r5,5).
value(r6,6).
value(r7,7).
value(r8,8).
value(r9,9).
value(rs,s).
value(rm,m).
value(rf,f).

movableTypes([1,2,3,4,5,6,7,8,s]).
movable(Piece):-
      value(Piece,PValue),
      movableTypes(MList),
      member(PValue,MList).



/*************************************************************
 ***                STRATEGO: MOVEMENT
 *************************************************************/
% ---------------------------------------
% define placement
precon(Player,place(Piece,From,To)):-
      player(Player),
      whosehand(Player,From),
      whosezone(Player,Zone),
      owner(Player,Piece),
      pieceAt(Piece,From),
      inzone(To,Zone),
```

```
      not(pieceAt(_,To)).
result(Player,place(Piece,From,To),
      [reveal(all,Piece,To,back),
      move(Piece,From,To),
      reveal(Player,Piece,To,front)]).


% ----------------------------------------
% define movement

% ---------
% -- ATTACK
precon(Player,amove(Piece,From,To)):-
      player(Player),
      whoseturn(Player),
      whosehand(Player,H),
      not(pieceAt(AnyPiece,H)),
      owner(Player,Piece),
      movable(Piece),
      pieceAt(Piece,From),
      pieceAt(EPiece,To),
      path(From,To),
      not(owner(Player,EPiece)).
% P = Spy, E equals 1.
result(Player,amove(Piece,From,To),
      [reveal(all,EPiece,To),
      remove(EPiece,To),
      move(Piece,From,To),
      reveal(all,Piece,To),
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            value(Piece,s),
            pieceAt(EPiece,To),
            value(EPiece,1).
% P = Spy, E equals Sply.
result(Player,amove(Piece,From,To),
      [remove(Piece,From),
      remove(EPiece,To),
      replace(Piece,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            value(Piece,s),
            pieceAt(EPiece,To),
            value(EPiece,s).
% P = Spy, E not 1.
result(Player,amove(Piece,From,To),
      [reveal(all,EPiece,To),
      reveal(all,Piece,From),
      remove(Piece,From),
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            value(Piece,s),
            pieceAt(EPiece,To),
```

```
            value(EPiece,PValue),
            PValue \= 1.
% E = Mine, P equals Miner.
result(Player,amove(Piece,From,To),
      [reveal(all,EPiece,To),
      remove(EPiece,To),
      move(Piece,From,To),
      reveal(all,Piece,To),
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            pieceAt(EPiece,To),
            value(Piece,8),
            value(EPiece,m).
% E = Mind, P not Miner.
result(Player,amove(Piece,From,To),
      [reveal(all,EPiece,To),
      reveal(all,Piece,To),
      remove(Piece,From),
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            pieceAt(EPiece,To),
            value(Piece,PValue),
            PValue \= 8,
            value(EPiece,m).
% E = Flag.
result(Player,amove(Piece,From,To),
      [reveal(all,EPiece,To),
      reveal(all,Piece,To),
      remove(EPiece,To),
      move(Piece,From,To),
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            pieceAt(EPiece,To),
            value(EPiece,f).
% P better than E.
result(Player,amove(Piece,From,To),
      [reveal(all,EPiece,To),
      reveal(all,Piece,To),
      remove(EPiece,To),
      move(Piece,From,To),
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            pieceAt(EPiece,To),
            value(Piece,PValue),
            value(EPiece,EValue),
            PValue > EValue.
% E better than P.
result(Player,amove(Piece,From,To),
      [reveal(all,EPiece,To),
      reveal(all,Piece,To),
      remove(Piece,From),
```

```
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            pieceAt(EPiece,To),
            value(Piece,PValue),
            value(EPiece,EValue),
            PValue < EValue.
% P equals E.
result(Player,amove(Piece,From,To),
      [reveal(all,EPiece,To),
      reveal(all,Piece,From),
      remove(Piece,From),
      remove(EPiece,To),
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]) :-
            nextPlayer(Player,Next),
            pieceAt(EPiece,To),
            value(Piece,PValue),
            value(EPiece,EValue),
            PValue = EValue.

% ----------------
% --- simple MOVE
precon(Player,move(Piece,From,To)):-
      player(Player),
      whoseturn(Player),
      whosehand(Player,H),
      not(pieceAt(AnyPiece,H)),
      owner(Player,Piece),
      movable(Piece),
      pieceAt(Piece,From),
      path(From,To),
      not(pieceAt(_,To)).
result(Player,move(Piece,From,To),
      [reveal(all,Piece,To,back),
      reveal(Player,Piece,To,front),
      move(Piece,From,To),
      replace(Player,whoseturn,Next),
      reveal(all,Piece,whoseturn)]):-
            nextPlayer(Player,Next).

% ----------------
% -- PASS
precon(Player,pass):-
      player(Player),
      not(whoseturn(Player)),
      whosehand(Player,H),
      not(pieceAt(AnyPiece,H)).
result(Player,pass,[]).


/********************** EOF ************************/
```

VITA

DAVID M. KAISER

2007	Doctoral Candidate in Computer Science, Florida International University
2000	Master's of Science Degree in Computer Science Florida International University
1994	Bachelor's of Science Degree in Computer Science Florida International University

PUBLICATIONS AND PRESENTATIONS

Kaiser, D. *Automatic Feature Extraction for Autonomous General Game Playing Agents*. Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems. AAMAS'07. May 14-18, 2007 Honolulu, Hawaii, USA

Kaiser, D. *The Design and Implementation of a Successful Autonomous Game Playing Agent*. Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference. FLAIRS 2007. May 7-9, Key West, Florida, USA