

# Automatic Feature Extraction for Autonomous General Game Playing Agents

David M. Kaiser  
School of Computer Science  
Florida International University  
Miami, FL  
david.kaiser@fiu.edu

## ABSTRACT

The General Game Playing (GGP) problem is concerned with developing systems capable of playing many different games, even games the system has never encountered before. Successful GGP agents must be able to extract relevant features from the formal game description and construct effective search heuristics. In this article, we present a procedure by which autonomous General Game Playing agents can generate effective and efficient search heuristics from the formal game description. The major aspect of our approach is an innovative technique to automatically extract critical features from the game structure. Our method has been incorporated into a fully implemented system that came in fourth place at the second General Game Playing Competition held at AAAI-06.

## Categories and Subject Descriptors

I.2.6 [Learning]: – *Knowledge acquisition*. I.2.8 [Problem Solving, Control Methods, and Search]: – *Heuristic methods*. I.2.11 [Distributed Artificial Intelligence]: – *Intelligent agents, Multiagent systems*.

## General Terms

Algorithms, Design, Experimentation, Theory.

## Keywords

Feature Extraction, General Game Playing.

## 1. INTRODUCTION

Specialized game playing systems are capable of beating the best human players in many games such as chess, checkers, Othello and backgammon. However, the spectacular achievements of these programs have not translated into success in more than a handful of problems. General Game Playing (GGP) is concerned with the development of systems that automate general cognitive processing technologies such as knowledge representation, reasoning and rational behavior. GGP agents are able to play previously unknown games, based exclusively on the description

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'07, May 14-18, 2007, Honolulu, Hawaii, USA.  
Copyright 2007 IFAAMAS.

of the game. Rather than relying on algorithms tuned for a specific game, they are able to adapt their behavior to each new game they encounter.

Feature Extraction is an important problem in many areas of computer science including machine learning, data mining, computer vision, bioinformatics and speech recognition [5]. Successful GGP agents must be able to extract pertinent features in each new game they are presented and adapt their behavior accordingly. This paper presents a method for constructing effective search heuristics for arbitrary games. The main contribution 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.

## 2. GENERAL GAME PLAYING

To compete effectively, game playing agents must make a series of moves that lead to a final winning position. However, the state space for most interesting games is too large to search exhaustively, so standard game-playing search techniques include some variant of the Min-Max algorithm with Alpha-Beta pruning [9]. 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.

To perform well, the evaluation function must be as accurate as possible. Systems designed to play specific games use manually selected features and optimization techniques such as opening books or end game databases to enhance the evaluation function. World Champion Chess Deep Blue [1] has an opening book of 4,000 positions and a summary of 700,000 grandmaster games. World Checkers Champion Chinook [12] has perfect information for over 443 billion end game positions.

Successful GGP agents, on the other hand, must be capable of playing many different games, even games they have never seen before. The central problem for a general game playing agent is to construct a heuristic evaluation function that performs efficiently for each different game it confronts. Therefore GGP systems must be able to perceive attributes in each new game and adapt their behavior accordingly without intervention of human control. 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.

### 3. RELATED WORK

The most relevant work to our own is that of Kuhlmann et al. [8]. 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.

HOYLE [2] is a system that can learn to play two-person, perfect information, finite board games. It uses game independent advisors, weighted for each particular game to improve its performance. Each advisor represents a different viewpoint on game playing, and takes a fairly narrow, but rational, view of the move selection problem. HOYLE quickly and efficiently identifies key information about the game but it seems to require a certain amount of hand crafting (i.e. programmer intervention) for each new game. HOYLE has learned to play Tic-tac-toe and Nine-men's Morris perfectly, but it is unclear how well HOYLE would play more complex games like chess.

The METAGAMER program [11] is a general game player that plays a subset of two-person, perfect information, deterministic games called symmetric chess-like games, which includes games like checkers and chess. Using features like piece count, piece mobility, threats, distance and material value, the METAGAMER system is able to generate effective evaluation functions for novel games within its domain.

WAR [6] 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 which are comparable in complexity to checkers and chess. The WAR system uses a genetic algorithm to learn each new game through self play. But unlike the method described here, WAR performs the learning process off-line, not as part of the game playing process.

### 4. THE ENVIRONMENT

In this paper, we focus our attention on the class of games described by the Game Description Language (GDL) and used within the Stanford General Game Playing framework [3]. GDL is a formal language for defining *deterministic* games with *perfect information*. 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. Games in which agents are not privy to the entire game state, such as Poker or Scrabble, are not perfect information games. A *deterministic* game is one in which the game states are decided entirely by the combined decisions of the competitors. Checkers is a deterministic game, but games involving rolling dice or shuffling cards are considered non-deterministic.

The Stanford GGP framework defines how participating agents compete. GGP agents are given the game description, their role within the game, the time limit available to analyze the game and

the time available to submit moves. Each player communicates with a central mediator, the *Gamemaster* through an HTTP connection. The *Gamemaster* transmits all game information to the players including the game description, start time, player moves and final game scores. Players communicate only with the *Gamemaster*, sending legal moves at the appropriate time. After each turn the *Gamemaster* informs each player of the moves made by each participant, until a terminal position is reached and the game ends. GGP agents must be fully autonomous without the intervention of human control.

```
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 1 - A toy game description in GDL.

The GDL language is a variant of first order logic based on the Knowledge Interchange Format (KIF) [4]. A toy game is described in Figure 1. Each GGP agent must be able to play any game, given such a description. In this example there is only one player (*role white*), one initial game state (*init (cell 1 1 b)*), and one legal move (*legal white (mark 1 1)*). The game is over when the player makes their 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.

### 5. FEATURE EXTRACTION

Feature identification is necessary whenever search cannot reach terminal game states from which to determine true payoffs, as is typical in most interesting games [13]. In a domain such as that considered in this paper, 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.

Although our system identifies many features from the game definition, we describe here only the agent’s identification of pieces. The game state from a game of “Chess” defined in GDL is used to illustrate our technique. Methods used to identify other features are largely different from this technique and their details are not discussed in this paper.

```
(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 2 - 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 2. In this example the first two arguments of the `cell` predicate represent coordinates and the third argument represents the contents of each location. 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”. 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.

```
(true (<predicate> <arg1> <arg2> <arg3>))
(true (cell a 1 wr))
```

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 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 3. The system needs a method to extract features that is more robust than simple pattern matching.

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

(true (place column1 row1 fortress))
(true (tyrant castle alpha prime))
(true (location noise primo keep first))
(true (position angrymuffin tower))
```

Figure 3 - Alternate game structure formats, before and after obfuscation.

## 6. OUR APPROACH

### 6.1 In General

In many types of 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`.

### 6.2 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> ... <argn>))`). 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:

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

However, the numerically stable algorithm in Figure 4, due to Knuth [7], who cites Welford [14], is used because it does not require storing the whole sequence of elements. Only the last value of the mean and the variance 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 4 - 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 a 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 checker 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.

### 6.3 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 5, 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)          cmp (step 2)          => (. 2)
(control white)  cmp (control black)
                                                         => (. black)
(cell a 1 wr)    cmp (cell a 1 wr) => (. . . .)
(cell a 2 wp)    cmp (cell a 2 b)  => (. . . b)
(cell a 3 b)     cmp (cell a 3 wp) => (. . . wp)

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

```

**Figure 5 - example of comparing two game states**

### 6.4 Piece Identification

The algorithm for extracting “piece” features from the game description is shown in Figure 6. The system first plays several random games and stores the results. The remaining calculations are all based on this recorded history.

```

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 gi in s
    Sort gi
    /* Detect motion */
    Compare gi with gi+1
  End For
End For

```

**Figure 6 - Algorithm for extracting game "piece" features.**

PieceInfo:

symid=10,	symbol=wr,	groupkey=1796,	argument=3,	symbolvariance=0.134054,	symbolmean=2.162162.
symid=12,	symbol=wp,	groupkey=1796,	argument=3,	symbolvariance=3.127927,	symbolmean=6.270270.
symid=19,	symbol=bp,	groupkey=1796,	argument=3,	symbolvariance=1.042162,	symbolmean=6.567565.
symid=21,	symbol=br,	groupkey=1796,	argument=3,	symbolvariance=0.0,	symbolmean=2.0.
symid=22,	symbol=wn,	groupkey=1796,	argument=3,	symbolvariance=0.243783,	symbolmean=1.445945.
symid=23,	symbol=bn,	groupkey=1796,	argument=3,	symbolvariance=0.0,	symbolmean=2.0.
symid=25,	symbol=wb,	groupkey=1796,	argument=3,	symbolvariance=0.211351,	symbolmean=1.689189.
symid=26,	symbol=bb,	groupkey=1796,	argument=3,	symbolvariance=0.038378,	symbolmean=2.040540.
symid=28,	symbol=wq,	groupkey=1796,	argument=3,	symbolvariance=0.0,	symbolmean=1.0.
symid=29,	symbol=bq,	groupkey=1796,	argument=3,	symbolvariance=0.0,	symbolmean=1.0.
symid=31,	symbol=wk,	groupkey=1796,	argument=3,	symbolvariance=0.142882,	symbolmean=1.175675.
symid=32,	symbol=bk,	groupkey=1796,	argument=3,	symbolvariance=0.0,	symbolmean=1.0.

Figure 7 – Features identified as pieces from the chess game description.

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 their 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.

Figure 7 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.

## 7. THE EVALUATION FUNCTION

Following the approach used in HOYLE [2] and WAR [6] 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 in our agent. It should be noted that this list is incomplete. Many important general heuristics are missing and the set will likely be expanded upon for future competitions. 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.

### 7.1 Game Structure Evaluators

The first group of evaluators generalizes concepts related to games that involve boards and pieces. The complicating factor with for 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 game 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.

### 7.2 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 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.

### 7.3 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 for self play. The agent first plays two games in which all players choose their moves at random. 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 whose purpose is to identify evaluators that are effective for the target game definition. For each unique piece type  $p$ , 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 random player. In order to play many games, the depth of look-ahead used in the search function is limited.

```

For every evaluator type  $t$ 
  For every piece type  $p$ 
    Construct Evaluator  $E(t,p)$ 
    Play game using  $E(t,p)$  vs. Random
    If  $E(t,p)$  wins then add  $E(t,p)$  to  $L$ 
  End For
End For

```

**Figure 8 – Algorithm to select piece evaluators to including in final evaluation function.**

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 3. 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.

## 8. THE COMPETITION

The AAI-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 top two scoring agents competed in the final championship match at the conference.

## 9. ASSESSMENT

Our system, OGRE, performed successfully during the entire three months of the competition, coming in fourth place out of twelve initial entrants [10]. 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. As shown in Table 2, our system won 34% of the matches it participated in.

**Table 2 - Results for games participated in during the AAI competition.**

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

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 and multi-player games.

With the aid of specially designed chess chips, Deep Blue is capable of examining over 200 million game states each second. And readily available publicly available chess playing programs such as GNUchess and Crafty can easily search through over 35,000 game states each second. On average our system is only capable of searching 100 game states each second. Thankfully, the effectiveness of the evaluation function generated by the system 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.

There are many other areas for investigation and improvement. One of our goals for future work is to improve the overall evaluation function. First, expanding the list of evaluators to include additional heuristics, both general and game specific, should improve performance. Second, using the outcome of matches against random players does not always provide enough information to determine the effectiveness of an evaluator and alternatives to this approach should be explored. Third, the system currently assigns similar weights to each evaluator, but it might prove beneficial to test different weighting strategies.

## 10. CONCLUSIONS

Feature Extraction is an important problem in many areas of computer science including machine learning, data mining, and computer vision. We have presented in this paper an innovative method for autonomous agents to extract key features from their knowledge of the environment. Our method has been fully implemented in an autonomous agent, called OGRE, which came in fourth place in the second AAAI General Game Playing competition. While procedures presented in this paper can lead to the development of better solutions for the general game playing problem, in the future we plan to extend our method of using variance to detect features in a broader problem space.

## 11. REFERENCES

- [1] Campbell, M., Hoane, A. J., Hsu, F. 2002. Deep Blue. *Artificial Intelligence* 134(1-2):57-83.
- [2] 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,: AAAI Press.
- [3] Genesereth, M., Love, N., and Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2): 62-72.
- [4] Genesereth, M. 1991. Knowledge interchange format. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Second Intl. Conference (KR'91)*.
- [5] Guyon, I. and Elisseeff, A. 2006. An Introduction to Feature Extraction. Feature Extraction, Foundations, and Applications. Series Studies in Fuzziness and Soft Computing, Physica-Verlag, Springer
- [6] Kaiser, D. 2000. A Generic Game Playing Machine, Master's thesis, Florida International University, Florida.
- [7] Knuth, D. E. 1998. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms, 3<sup>rd</sup> edn., p 232. Boston: AddisonWesley.
- [8] Kuhlmann, G., Dresner, K., and Stone, P. 2006. Automatic Heuristic Construction in a Complete General Game Player. In Proceedings of the Twenty First National Conference on Artificial Intelligence. p 1457-62.
- [9] Levy, D. N. L. and Newborn, M. M., 1991. *How Computers Play Chess*. W.H. Freeman and Company.
- [10] Love, N., 2006. General Game Playing Competition Results [<http://games.stanford.edu/2006results.html>] accessed September 24, 2006.
- [11] Pell, B. 1993. Strategy Generation and Evaluation for Meta-Game Playing. PhD thesis, University of Cambridge.
- [12] Schaeffer, J., Treloar, N., Lu, P., and Bryant, M., 1994. Chinook: The Man-Machine World Checkers Champion. *AI Magazine*. 17(1): 21-29.
- [13] Utgoff, P.E. 2001. Feature construction for game playing. In Fuerenkranz & Kubat (Eds.), *Machines that learn to play games*. P 131-152. Nova Science Publishers.
- [14] Welford, B.P. 1962. Note on a method for calculating sums of squares and products. *Technometrics* 4(3):419-420.