

# The Design and Implementation of a Successful General Game Playing Agent

David M. Kaiser

Florida International University  
11200 S.W. 8th Street  
Miami, Florida 33199  
dkaiser@mdc.edu

## Abstract

General Game Playing is the problem of designing an agent capable of playing any previously unknown game when given only the rules. This paper describes the implementation architecture and design issues behind OGRE, a General Game Playing agent designed at Florida International University. Our main contribution is an innovative algorithm for automatically generating efficient evaluation functions for previously unfamiliar games. The system competed successfully at the second AAAI General Game Playing competition held at AAAI-06.

## Introduction

General Game Playing (GGP) is concerned with the development and use of systems that automate general cognitive processing technologies (such as knowledge representation, reasoning and rational behavior) (Genesereth, Love and Pell 2005). Current 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 in these areas have not translated into success in more than a handful of problems. Therefore, key concerns in the development of more powerful GGP systems are to provide an ability to solve a large range of problems, and to provide the ability to solve more difficult problems within the same resource limits.

This paper presents the design and implementation of OGRE, a fully autonomous agent created to participate in the second AAAI General Game Playing competition which was held at the AAAI 2006 in Boston. The main contribution is an original method for automatically constructing effective search heuristics based solely on the game description. The system is fully implemented and competed successfully at the AAAI-06 General Game Playing competition. OGRE came in fourth place out of twelve initial participants.

## General Game Player Design Issues

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* (Levy and Newborn 1991). 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 optimization techniques such as opening books or end game databases to enhance the evaluation function. World Champion Chess 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, et al. 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.

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. Even if the agent 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.

## Related Work

HOYLE (Epstein 1994) 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 games 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 (Pell 1993) 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.

The GGP system developed at the University of Texas (Kuhlmann, Dresner and Stone 2005) competed along with Ogre's predecessor, Goblin, in the first GGP competition. The UTexas system was intended to compete in the same domain as Ogre, so it is not surprising that they developed some solutions to common problems that are quite similar to ours, such as successor function identification. They also provided interesting solutions to problems not addressed by the Ogre design such as identifying teammates in multi-player games.

Ogre relies heavily on two of our previous system designs: Goblin and WAR. Goblin took second place during the first AAI General Game Playing competition at AAI-05. WAR (Kaiser 2000) is system 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. Ogre and WAR do not play the same class of games, but Ogre builds on the notion of using a set of modules to evaluate separate general game features.

## The Design of Ogre

Ogre is a GGP agent designed to play any game defined in the Game Description Language (GDL) and compete within the Stanford GGP framework. The language is a subset of first-order logic based on KIF (Genesereth 1991).

GDL is a formal language for defining *deterministic* games with *perfect information*. A game, in which all the players have complete information of the current game state, is called a *perfect information* game. 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 nondeterministic.

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 submit 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.

## Feature Extraction

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, an instance of the eight puzzle game state a) (box 1 b 2 3 4 5 6 7 8) (turn 14) and another game state b) (box 1 b 2 3 4 5 6 7 8) (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.

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:

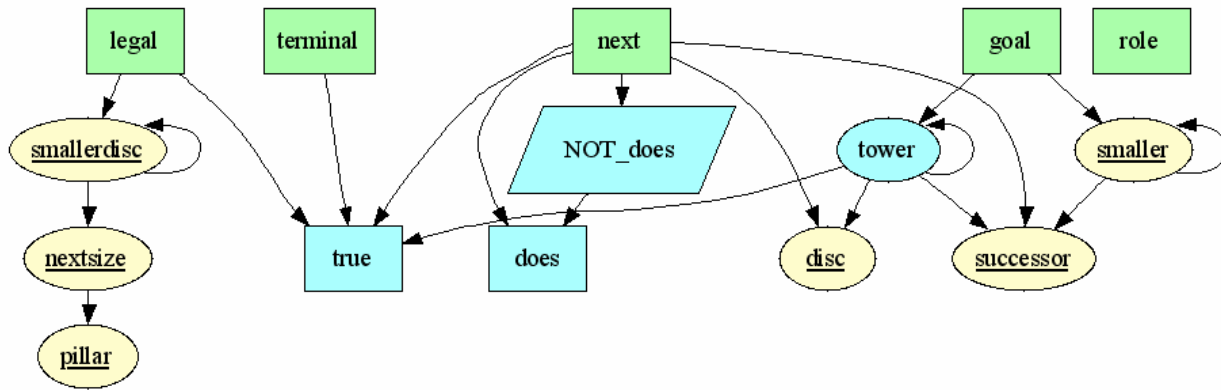
```
(<successor> <value0> <value1>
<successor> <value1> <value2>
<successor> <value2> <value3>
. . .
<successor> <valueN-1> <valueN>)
```

Where <successor> can be any relation constant, and the <value> components are object constants.

It is then possible to identify predicate functions with the following format as possible turn counters.

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



**Figure 1** – Dependency graph for predicates in the game Towers of Hanoi. Rectangles represent reserved GDL predicates. Ovals represent game specific predicates. Parallelograms indicate negated predicates. Static predicates are underlined.

game descriptions that were available during the development of these systems.

Unfortunately, the method is quite brittle. Encoding the turn counter differently prevents recognition of this feature.

### Identify Static Clauses

Ogre also creates a dependency graph of the predicates in the game description. The dependency graph identifies clauses that are static. Figure 1 shows a dependency graph generated by the system.

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.

### The Evaluation Function

Following the approach used in 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. 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.

**Game Structure Evaluators** The first group of evaluators generalizes concepts related to games that involve boards and pieces. The complicating factor 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. 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.

**Game Definition Evaluators** The second group of evaluators do not rely on information derived from the structure of the game. These evaluators encapsulate very general heuristics that are applicable to a very 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.

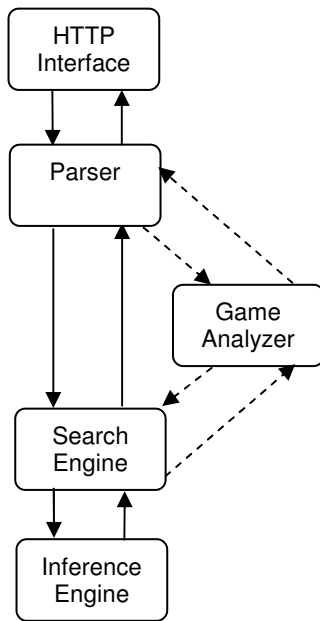
**Pattern:** compares the current game state with a pattern found in the goal state. Helps solve several simple puzzles quickly.

**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.

The system combines these evaluators into a single evaluation function by playing games internally against a player that makes random moves. This method is described in more detail in the next section.

## The Implementation of Ogre

Ogre is written entirely in Java. The system consists of five key components: HTTP Interface, Parser, Game Analyzer, Search Engine and Inference Engine. The overall process architecture of the Ogre implementation is shown in Figure 2.



**Figure 2** – Ogre architecture. The Game Analyzer is used during the analysis phase before the first turn is submitted.

### HTTP Interface

The Ogre HTTP interface is a very simple HTTP server. The Stanford GGP framework 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.

### 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 forward-chaining inference engine developed at Stanford (Fikes, Frank, and Jenkins 2003). In addition to the parser, Ogre’s ancestor, Goblin, used the JTP inference engine 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.

### 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.

**Constructing the Evaluation Function** The system uses approximately 50% of the time given in the analysis phase for self play. The system 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.

The algorithm used to select evaluators which involve pieces is outlined in Figure 3.

```
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 3** – Algorithm for selecting evaluators.

Every evaluator returns a positive number. The final evaluation function is the sum of values returned by all the selected evaluators. One instance of evaluator is created for each piece identified in the game definition.

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, Ogre assigns similar weights to each evaluator, but it might prove beneficial to test different weighting strategies.

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.

## 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. In practice, this is unlikely, 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* (Russel and Norvig 2003) 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.

## 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 entirely new inference engine. It is significantly faster than the one used by Goblin and includes some GDL specific enhancements. The basic inference algorithm is shown in Figure 4.

The most significant enhancement is the *static predicate* cache. As described earlier, *static predicates* are those predicates that are not dependent on the reserved GDL predicates TRUE or DOES. The Ogre inference engine caches the results of any resolution done on a clause containing a *static 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  $\theta$  THEN
      G2 = Unify(L, G1,  $\theta$ )
      Push right-literals of G2 onto G
    ELSE
      IF backtrack() fails
        THEN return FALSE
    ENDIF
  ENDFOR
}
```

**Figure 4** – Inference Algorithm. The queryStatic() function stores and retrieves literals to/from the cache.

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 none 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.

## Assessment

Ogre performed very successfully in the AAI-06 competition (Love 2006), coming in fourth out of the initial twelve entrants. There were eleven days of competition held over the course of three months, culminating in a final match on the third day of the conference. Each system 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 one time, with players switching sides.

		One-player Points		Two-player Points		Multi player Points	
Fluxplayer	1st	1520	80%	2792	59%	350	50%
Clunepayer	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	

**Table 1** – The un-weighted points acquired by the top four players. Each player had the opportunity to earn 7300 points.

Each match afforded players the opportunity to acquire up to 100 points. As shown in table 1, Ogre performed best in games involving more than two players and worst in puzzles. Ogre received only 43% of the possible points from puzzles compared to the 1<sup>st</sup> 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 Ogre is only able to search through 100 game states each second. However, the effectiveness of the evaluation functions generated by the system seems to compensate for this serious shortfall.

## Conclusion & Future Work

In this paper, we presented the implementation architecture and design issues behind Ogre, a General Game Playing agent. We described an innovative methodology for generating specialized evaluation functions for previously unfamiliar games.

While the system performed well during competition, it is clear that there are many areas that could be improved and much work remains to be done. In addition to new evaluators and better feature detection the area of overall system speed should be addressed.

The core inference engine could be made faster. Because every fact in each game state must be inferred from the previous state combine with each player’s moves, a significant percentage of the system’s computation time is taken performing resolution. While the amount of time is generally dependent on the game description, it is clear that improvements in this component would improve the entire systems functionality.

## References

- Astrachan, O.L., and Stickel, M.E., 1992. Caching and Lemmaizing in Model Elimination Theorem Provers, In *Proceedings of Automated Deduction - CADE-11, 11th International Conference on Automated Deduction*, 224-238.
- Campbell, M., Hoane, A. J., Hsu, F. 2002. Deep Blue. *Artificial Intelligence* 134(1-2):57-83.
- 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.
- Fikes R., Frank, G., and 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. July 27-30.
- Genesereth, M., Love, N., and Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2): 62-72.
- Kaiser, D. 2000. A Generic Game Playing Machine, Master’s thesis, Florida International University, Florida.
- 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.
- Levy, D. N. L. and Newborn, M. M., 1991. *How Computers Play Chess*. W.H. Freeman and Company.
- Love, N., 2006 General Game Playing Competition Results [<http://games.stanford.edu/2006results.html>] accessed September 24, 2006.
- Pell, B. 1993. Strategy Generation and Evaluation for Meta-Game Playing. PhD thesis, University of Cambridge.
- Russel, S. and Norvig, P. 2003. *Artificial Intelligence, A Modern Approach (Second Edition)*, Prentice Hall International, Inc.
- Sakuta, M. and Iida, H. 2000. *Solving Kriegspiel-Like Problems: Exploiting a Transposition Table*, ICGA JOURNAL, 23(4): 218-229.
- Schaeffer, J., Treloar, N., Lu, P., and Bryant, M., 1994. Chinook: The Man-Machine World Checkers Champion. *AI Magazine*. 17(1): 21-29.
- Sturtevant, N. R., and 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*, 201-207.