Automatic Heuristic Construction in a Complete General Game Player*

Gregory Kuhlmann, Kurt Dresner and Peter Stone
Department of Computer Sciences, The University of Texas at Austin
1 University Station C0500, Austin, Texas 78712-1188
{kuhlmann,kdresner,pstone}@cs.utexas.edu

Abstract

Computer game players are typically designed to play a single game: today's best chess-playing programs cannot play checkers, or even tic-tac-toe. *General* Game Playing is the problem of designing an agent capable of playing many different previously unseen games. The first AAAI General Game Playing Competition was held at AAAI 2005 in order to promote research in this area. In this article, we survey some of the issues involved in creating a general game playing system and introduce our entry to that event. The main feature of our approach is a novel method for automatically constructing effective search heuristics based on the formal game description. Our agent is fully implemented and tested in a range of different games.

Introduction

Creating programs that can play games such as chess, checkers, and backgammon, at a high level has long been a challenge and benchmark for AI. While several game-playing systems developed in the past, such as Deep Blue (Campbell, Jr., & Hsu 2002), Chinook (Schaeffer *et al.* 1992), and TD-gammon (Tesauro 1994) have demonstrated competitive play against human players, such systems are limited in that they play only one particular game and they typically must be supplied with game-specific knowledge. While their performance is impressive, it is difficult to to determine if their success is due to the particular game-playing technique or due to the human game analysis.

A *general* game playing agent must be able to take as input a description of a game's rules and proceed to play without any human input. Doing so requires the integration of several AI components, including theorem proving, feature discovery, heuristic search, and potentially learning.

This paper presents a complete and fully autonomous general game playing agent designed to participate in the first AAAI General Game Playing (GGP) Competition which was held at the AAAI 2005 in Pittsburgh (Genesereth & Love 2005). The main contribution is a novel method for automatically constructing effective search heuristics based on the formal game description. Our agent is fully implemented and tested on several different games.

General Game Playing

The General Game Playing problem is the challenge of creating a system capable of playing games with which it has had no prior experience. The definition of a "game" can be quite broad, ranging from single-state matrix games such as the Prisoner's Dilemma to complex, dynamic tasks like robot soccer. A general game playing scenario is specified by three main components: (1) the class of games to be considered, (2) the domain knowledge prior to the start of the game, and (3) the performance measure.

First, in this paper, we restrict our attention to the class of games that were considered in the 2005 AAAI GGP competition, namely discrete state, deterministic, perfect information games. The games may be single or multi-player and they may be turn-taking or simultaneous decision. By *deterministic*, we mean that given a state of the game, and a joint set of actions taken by all players, the next state is uniquely determined. Go and Othello are examples of games from this class. However, Backgammon is not, because dice rolls are nondeterministic. In a *perfect information* game, the complete state of the game is known by all participants. Chess and Checkers qualify as perfect information games, because the state of the game is completely captured by the positions of the pieces on the board, which is in plain sight. In contrast, games such as Poker and Battleship do not qualify as perfect information games because players hide part of the game state from their opponents.

Second, in addition to the set of games to be considered, a general game playing scenario is parameterized by the amount and type of domain knowledge given to the players prior to the start of the game. For example, in the Learning Machine Challenge sponsored by Ai in January of 2002, the players were told nothing more than the set of legal moves. In the scenario adopted in this work, players are given a formal description of the rules of the game. For our purposes, the game description, at a minimum, must contain sufficient information to allow the agent to simulate the game on its own. Our approach can take advantage of certain kinds of structure in the game description, as we will demonstrate.

Third, we must specify how agent performance is to be measured. In our setup, an agent is evaluated based on the number of points earned in a single shot game against competing agents. The identities of the opponents are hidden.

The key question that our work seeks to address is: In the general game playing scenario described above, how can we leverage the formal game description to improve agent performance? Before we answer this question, we will describe one concrete instance of this scenario, namely the AAAI GGP competition. The AAAI GGP scenario was the main motivating scenario for our agent development and is the scenario in which all of our empirical results are reported.

*Supported in part by NSF CAREER award IIS-0237699.

The AAAI GGP Competition

The first annual AAAI General Game Playing Competition was held at the 2005 AAAI conference in Pittsburgh (Genesereth & Love 2005). Nine teams participated in that event. In each of three main rounds, game playing agents were divided into groups to compete in both two- and three-player games. The games included a three player version of Chinese Checkers, a variant of Othello called "Nothello", and in the final round, a simultaneous decision racing game called "Racetrack Corridor." The complete details of the competition are available online[1].

In the competition setup, each player runs as an independent server process. At the start of a game, a process called the *Game Manager* connects to each player and sends the game description along with time limits called the *start clock* and *play clock*. Players have the duration of the start clock to analyze the game description before the game begins. Each turn, players get the duration of the play clock to choose and announce their moves. After each turn, the Game Manager informs the players of the moves made by each player. The game continues until a terminal state is reached. No human intervention is permitted at any point: the general game player must be a complete and fully autonomous agent.

Game Description Language

The Game Description Language (GDL), used in the competition, is a first-order logical description language based on KIF (Genesereth 1991). In GDL, games are modeled as state machines in which a state is the set of true facts at a given time. Using a theorem prover, an agent can derive its legal moves, the next state given the moves of all players, and whether or not it has won. Each of these operations requires theorem proving; simulating games can be costly. One of the research challenges of GGP is to find efficient methods for reasoning about games described in first-order languages. Part of the description for a game called "Minichess" is shown in Figure 1. A GGP agent must be able to play any game, given such a description. We illustrate the features of GDL through this example.

First, GDL declares the game's roles (line 1). The Game Manager assigns each game playing agent a role to play. "Minichess" has two roles, `white` and `black`, making it a two player game. Next, the initial state of the game is defined (2–7). Each functional term inside an `init` relation is true in the initial state. Besides `init`, none of the tokens in these lines are GDL keywords. The predicates `cell`, `control` and `step` are all game-specific. Even the numbers do not have any external meaning. If any of these tokens were to be replaced by a different token throughout the description, the meaning would not change. This lack of commitment to any lexical semantics will be important when we discuss our approach to feature discovery.

GDL also defines the set of legal moves available to each role through `legal` rules (8–15). The <= symbol is the reverse implication operator. Tokens beginning with a question mark are variables. The `true` relation is affirmative if its argument can be satisfied in the current state. Each player must have at least one legal move in each nonterminal state for the game to be valid. The second rule (14–15) demonstrates how turn-taking is simulated in GDL by requiring a player to execute a null action in certain states.

```
1.  (role white) (role black)
2.  (init (cell a 1 b))  (init (cell a 2 b))
3.  (init (cell a 3 b))  (init (cell a 4 bk))
...
4.  (init (cell d 1 wr)) (init (cell d 2 b))
5.  (init (cell d 3 b))  (init (cell d 4 b))
6.  (init (control white))
7.  (init (step 1))
...
8.  (<= (legal white (move wk ?u ?v ?x ?y))
9.      (true (control white))
10.     (true (cell ?u ?v wk))
11.     (kingmove ?u ?v ?x ?y)
12.     (true (cell ?x ?y b))
13.     (not (restricted ?x ?y)))
14. (<= (legal white noop)
15.     (true (control black)))
...
16. (<= (next (cell ?x ?y ?p))
17.     (does ?player (move ?p ?u ?v ?x ?y)))
18. (<= (next (step ?y))
19.     (true (step ?x))
20.     (succ ?x ?y))
...
21. (succ 1 2) (succ 2 3) ... (succ 9 10)
22. (nextcol a b) (nextcol b c) (nextcol c d)
...
23. (<= (goal white 100)
24.     checkmate)
25. (<= (goal black 100)
26.     (not checkmate))
...
27. (<= terminal
28.     (true (step 10)))
29. (<= terminal
30.     stuck)
```

Figure 1: Partial game description for "Minichess", GDL keywords shown in **bold**.

The state transition function is defined using the `next` keyword (16–20). Transition rules are used to find the next state, given the current state and the actions of all players. The `does` predicate is true if the given player selected the given action in the current state. Finally, GDL defines rules to determine when the game state is `terminal` (27–30). When the game ends, each player receives the reward defined by the game's `goal` rules (23–26).

A game description may define additional relations to simplify the conditions of other rules and support numerical relationships. For instance, the `succ` relation (21) defines how the game's step counter is incremented, and the `nextcol` relation (22) orders the columns of the chess board. As we will discuss, identifying these kinds of numerical relationships is extremely valuable, as they serve as a bridge between logical and numerical representations.

[1] `http://games.stanford.edu/results.html`

Approach

In our game playing scenario, in which an agent may look ahead by simulating moves, an obvious choice of approach is search. Most existing game playing systems for the types of game that we consider are based upon the Minimax search algorithm. Well-known examples include Chinook (Schaeffer *et al.* 1992) for Checkers and Deep Blue (Campbell, Jr., & Hsu 2002) for Chess. Even learning-based systems such as TD-gammon (Tesauro 1994) incorporate search for action selection. However, unless the state space is small enough to be searched exhaustively, the agent must use a heuristic evaluation function to evaluate non-terminal nodes and thus bound search depth. Such evaluation functions are highly game-specific, and much of the human effort in developing game playing systems is spent on manually tuning them.

In a general game playing setting, the evaluation function cannot be provided by a human. Because each lookahead requires costly theorem proving, exhaustive search is not a viable option. To make reasoning as efficient as we could, our agent uses a Prolog-style interpreter for theorem proving. In early testing, we found that this was significantly faster than other kinds of resolution. Even so, all but the smallest game trees are beyond the reach of exhaustive search.

To overcome these issues, we developed a method for generating heuristic evaluation functions automatically. We construct heuristics from features identified in the game description. Candidate heuristics are evaluated in parallel during action selection to find the best move. We discuss the details of our search algorithm in the next section before moving on to the heuristic construction algorithm.

Search Algorithm

The search algorithm employed by our player is based on alpha-beta pruning (Knuth & Moore 1975). Without a heuristic, the terminal nodes of the game tree are visited in a depth-first search order. For heuristic search, we use iterative deepening (Korf 1985) to bound the search depth. We include two general-purpose enhancements: transposition tables, which cache the value of states encountered previously in the search; and the history heuristic, which reorders children based on their values found in lower-depth searches. Other general techniques exist, but the combination of these two enhancements accounts for most of the search space cutoff when combined with additional techniques (Schaeffer 1989). We extended this basic minimax search algorithm to allow simultaneous decision games and games with more than two players.

Although our implementation was designed to work for the broadest possible set of games, we made one important simplifying assumption. We assume that each player in the game can be placed into one of two sets: teammates and opponents. Thus, every player is either with us or against us. Our simple but strict definition of a teammate is a player that always receives the same reward that we do. We determine which team a player is on through internal simulation (see below). By treating somewhat cooperative players as opponents, we avoid the need to maintain a separate utility function for each team, which can be expensive since standard alpha-beta pruning can no longer be used.

Our player uses the same search procedure for turn-taking and simultaneous decision games. The procedure also applies to games with a mix of the two. At a turn-taking node, if the move is being made by a teammate, it is treated as a maximization node. If it is an opponent's turn to move, it is a minimization node. This type of search is an instance of the *paranoid algorithm* (Sturtevant & Korf 2000). If it is a simultaneous move node, because of the unpredictability of our opponents, we assume that our opponents choose their moves uniformly at random. We choose our action from the joint action of our team that maximizes our expected reward, based on that assumption. If we have prior knowledge of the opponent, either from past games or earlier moves in the current game, opponent modeling methods could be applied at this point, as could game theoretic reasoning.

Identifying Syntactic Structures

The first step toward constructing a heuristic evaluation function is to identify useful structures in the game description. Useful structures include time counters, game boards, movable pieces, commodities, etc. Our agent identifies these structures by syntactically matching game description clauses to a set of template patterns. Although the syntax of these templates are specific to GDL, they are conceptually general, and the template matching procedure could be applied to other languages. Still, we include the idiosyncrasies of GDL in our discussion both to make the procedure concrete and to aid future competition participants. Specifically, we describe our agent's identification of five structural game elements: successor relations, counters, boards, markers, pieces, and quantities.

As mentioned above, other than keywords, GDL tokens have no pre-defined meaning to the player. In fact, during the competition, the tokens were scrambled to prevent the use of lexical clues. Therefore, all structures are detected by their syntactic structure alone. For example, one of the most basic structures our agent looks for is a *successor* relation. This type of relation induces a total ordering over a set of objects. We have already seen two examples in lines 21–22 of Figure 1. In a competition setting, the same successor relations may be look something like the following:

```
(tcsh pico cpio) (tcsh cpio grep) ... (tcsh echo ping)
(quiet q w) (quiet w e) (quiet e r) (quiet r t)
```

Our system can still identify these relations as successors because order is a structural, rather than lexical, property. When a successor relation is detected, the objects in its domain can be compared to one another and, in some cases, incremented, decremented, added and subtracted. Identifying such structures is very important. If, for example, the agent could not identify them in "Minichess," it would have no way to know that step 8 comes after step 7 or that column a is next to column b, both of which are useful.

A game description may include several successor relations, which the agent uses to identify additional structures such as a counter. A counter is a functional term in the game's state that increments each time step. Our agent identifies it by looking for a rule that matches the template:

```
(<= (next (<counter> ?<var1>))
    (true (<counter> ?<var2>))
    (<successor> ?<var2> ?<var1>))
```

where <counter> is the identified function, and <successor> is some successor relation. The order of the antecedents is not important. The step function in "Minichess" is an example of a time step counter, as can be seen in lines 18–20 of Figure 1.

Our game player identifies counters for several reasons. First, if the counter terminates the game in a fixed number of steps, it may be removed during internal simulation to increase the chances of encountering a goal state. More importantly, in some games, leaving the counter in the state representation causes state aliasing, making search inefficient. Therefore, the function is removed before caching a state's value in the transposition table.

Another example of a structure that our player attempts to identify is a *board*. A board is a two dimensional grid of cells that change state, such as a chess or checkers board. When the agent receives the game description, it assumes every ternary function in the state is a board. However, one of the board's arguments must correspond to the cell's state, which can never have two different values simultaneously — two of its arguments are inputs and the third is an output. We verify this property using internal simulation.

According to our definition of a board, a candidate is rejected if it ever has two objects in the same place. Although such a board may be valid, we choose to eliminate it to prevent false positives. "Minichess" has one board: the cell function. If a board's input arguments are ordered by some successor relation, then we say that the board is *ordered*.

Once a board is identified, our player looks for additional related structures such as *markers*, which are objects that occupy the cells of the board. If a marker is in only one cell at a time, we distinguish it as a *piece*. For example, in "Minichess", the white rook, wr, and the black king bk are pieces. However, games like Othello where, for instance, a black object may be in multiple places, have only markers.

Our player also identifies additional structures that do not involve boards or pieces. These structures are functions in the state that may quantify an amount of something, such as money in Monopoly. These are identified as those relations having ordered output arguments. We discuss the distinction between input and output arguments in the next section.

Internal Simulation

We have mentioned several situations in which we needed to prove an invariant about states of the game. For instance, we need to prove an invariant about terminal states to divide the game's roles into teams. We also need to prove invariants about the input and output modes of relations to identify boards, markers and pieces. Rather than proving these invariants formally, which would be time-consuming, our agent uses an approximate method to become reasonably certain that they hold. The agent uses its theorem prover to simulate a game of random players, during which, the agent checks whether or not its currently hypothesized invariants hold. To demonstrate, we trace the process of identifying the board and pieces in the "Minichess" example.

The agent assumes that cell is a board because it is the only ternary function in the state. From just checking the initial state, the agent is able to determine that the only candidate mode is (cell + + −), where + signifies an input argument and − signifies an output argument. This turns out to be the right answer and no further refinement is necessary.

Our agent also initially assumes that every object appearing in the output argument of the function is a piece. Therefore, in the example, wr, bk and b are all initially pieces. From checking the initial state, the agent eliminates b as a piece because it is in more than one place at a time. Further simulation of the game never rejects wr or bk as pieces.

As for the teams, our player assumes that black and white are on the same team. Once the first terminal state is reached during simulation, though, this hypothesis is rejected. "Minichess" is a zero sum game without any conditions for a tie, and therefore would never assign equal reward to the two roles. After the first simulated game, black and white are separated into two teams.

"Minichess" is an easy case in that the correct hypotheses are reached very quickly. Other games may require more simulation. In the games that we have encountered thus far, however, teams were identified after at most 2 games and boards and pieces stabilized after roughly 3 time steps.

In the competition, our agent ran the internal simulation for the first 10% of the start clock, with a minimum of 50 states visited. For a typical game, this amount of simulation was more than sufficient to establish a high degree of confidence in the agent's hypotheses.

From Syntactic Structures to Features

The structures identified in the previously described processes suggest several interesting features. We use *feature* to mean a numerical value, calculated from the game state, that has potential to be correlated with the outcome of the game. If the dimensions of the board are ordered, then our agent computes the x and y coordinates of each piece as the natural numbers associated with the input arguments' indices in their corresponding successor relations. For example, the coordinates of wr in the initial state of the "Minichess" example are $(3, 0)$. From these coordinates, the agent can also calculate the Manhattan distances between pieces.

If a board is not ordered, it may still produce useful features, including the number of occurrences of each marker. In addition, the agent generates features corresponding to the values of each quantifiable amount. A complete list of features generated by the player is shown in Table 1.

Identified Structure	Generated Features
Ordered Board w/ Pieces	Each piece's X coordinate Each piece's Y coordinate Manhattan distance between each pair of pieces Sum of pair-wise Manhattan distances
Board w/o Pieces	Number of markers of each type
Quantity	Amount

Table 1: Generated features for identified structures.

From Features to Heuristics

From the set of generated features, our game player creates heuristics to guide search. In traditional single-game-playing systems, multiple features are manually weighted

and combined to create a single evaluation function. Because the games to be played are not known to the designer, that option is not available to agents in the general game playing scenario. While it may be possible to learn the evaluation function, as is done by TD-gammon, doing this efficiently in the general case remains an open problem.

Instead of constructing a single heuristic function, our agent constructs a set of candidate heuristics, each being the maximization or minimization of a single feature. By including both, the agent can handle games with counterintuitive goal conditions. As it turned out, "Nothello" gave us a great example of such a game during the competition. In "Nothello", the player with the fewest markers at the end of the game wins. Because the agent generates a heuristic to minimize the number of its own markers, it had a candidate heuristic that matched well with the game's goal.

We implement the candidate heuristics as board evaluation functions that linearly map the feature's value to an expected reward between $R^- + 1$ and $R^+ - 1$, where R^- and R^+ are the minimum and maximum goal values achievable by the player, as described by the `goal` predicate. The values of the maximizing and minimizing heuristic functions are calculated respectively as follows:

$$H(s) = 1 + R^- + (R^+ - R^- - 2) * V(s)$$
$$H(s) = 1 + R^- + (R^+ - R^- - 2) * [1 - V(s)]$$

where $H(s)$ is the value of heuristic H in state s and $V(s)$ is the value of the feature, scaled from 0 to 1. We scale the heuristic function in this way so that a definite loss is always worse than any approximated value, and likewise, a definite win is always better than an unknown outcome.

Distributed Search

Not all of the heuristics that the player generates from the game description are particularly good for that game. Nor is the best choice of heuristic necessarily the same throughout the course of the game. Rather than attempting to extract hints about the goal from the goal conditions in the game description, which could be arbitrarily complex, we evaluate the candidate heuristics online using distributed search.

In addition to the main game player (GP) process, our player launches several remote slave (Slv) processes. In each play clock, the main process informs each slave of the current state, s, and assigns each one a heuristic, h, to use for search. Occasionally, the slave processes respond with their best action so far, a_i. Before the play clock expires, the game player evaluates the suggested actions, chooses the "best" one, a^*, and sends it to the game manager (GM). This procedure, with one slave, is illustrated in Figure 2.

The same heuristic may be assigned to multiple slaves if the number of heuristics is smaller than the number of slaves. This was typically the case during the competition, with roughly two dozen heuristics and almost 200 slaves. Although this redundancy typically leads to significant duplicated effort, ties between action scores are broken randomly during search and thus the different processes end up exploring different parts of the search space. Also, several slaves perform exhaustive search using no heuristic. This ensures optimal play near the end of the game.

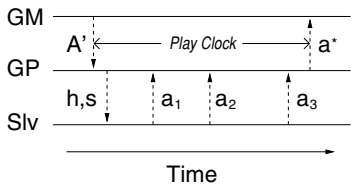

Figure 2: Messages between the Game Manager, Game Player, and a Slave process during a single play clock.

Choosing from among the suggested actions is a significant challenge, but one component of the strategy is clear: if exhaustive search returns an action, the agent should always choose it — the action is guaranteed to be optimal. Beyond that, our agent prefers actions from deeper searches to those from shallower ones, as deeper searches allow the agent to better evaluate the long term effects of its actions. It is tricky, however, to choose between actions nominated by different heuristics. Doing so in a principled way, perhaps by taking into account the game's goal condition, remains an open challenge and is an important direction for future work.

Experimental Results

After a successful start enabling our agent to reach the finals (3 teams out of the original 9), a networking glitch blocking connectivity between the competition site and our servers at home prevented our full participation in the finals. In this section, we present controlled experiments isolating the main contributions of this paper, namely the feature discovery and automatic heuristic construction methods.

Without the feature discovery and automatic heuristic construction techniques described above, a game player could only resort to performing exhaustive search for the entire game. While this strategy results in optimal play near the end of the game, the consequences of mistakes made early on would likely be irreparable. To measure the impact of our contributions on the player's performance, we conducted experiments in which a player with a generated heuristic competes with players performing exhaustive search.

To isolate the feature discovery and heuristic construction processes from the heuristic selection method, we do not use distributed search to evaluate heuristics. Instead, we choose a single heuristic for the player to use. Doing so emulates what we would expect from an agent with our heuristic construction method, but with a better heuristic evaluation method. In all cases, we choose the heuristic from preexisting options based on our knowledge of the game, but without revision or experimentation after the initial choice.

The exhaustive and heuristic players were pitted against each other in 3 different games. The first game, "Nothello", which we introduced before, is a variant of Othello in which each of the two players tries to end the game with fewer markers than their opponent. A win earns a player 100 points and a draw is worth 50. The generated heuristic that we chose was the one that minimized the number of markers for our player. We hypothesized that minimizing a player's markers in the short term could be a win in the long run.

The second game, "Hallway", is a two player game played on a chess board. Each player controls a pawn that, starting on opposite sides, must reach the other side before the op-

ponent to earn 100 points. During the game, a player may place up to four walls to hinder their opponent's progress. If neither player reaches the other side in 200 time steps, then the game is a draw and each player receives 50 points. The heuristic we chose for "Hallway" maximized the x-coordinate of our player's piece. This heuristic encourages the player to move closer to the opposite side of the board.

Lastly, "Farmers" is a three player commodities trading game. The three agents each start with equal amounts of money, and may use it to buy and sell cotton, cloth, wheat and flour. By saving up for a farm or factory, a player may produce additional commodities. Each time step, all players make their trading decisions simultaneously. When the game ends ten steps later, the player with the most money wins. If there is a tie, all players with the most money receive 100 points. The heuristic that was chosen for this game was the one that maximizes the player's own money. Both of the heuristic player's opponents used exhaustive search.

All three games were developed by the organizers of the AAAI GGP competition and were not created to match the constructed heuristics. For each game, we ran several matches with start and play clocks of 60 seconds. We recorded the number of games in which the agent using the generated heuristic scored 100 points. The results are shown in Table 2. In "Nothello", our agent won all 15 of its matches. The probability, p, that a random player would perform as well as our agent is roughly 10^{-5}. We calculated this based on the prior probability of winning using random moves, which was not quite 50% due to a small chance of ties. We found this probability experimentally by running 500 matches with all players choosing random moves. The p values for the remaining games were found similarly.

Game	Wins	Matches	p
Nothello	15	15	10^{-5}
Hallway	15	15	10^{-11}
Farmers	11	25	0.234

Table 2: Results for the agent using a generated heuristic versus one or more players using exhaustive search.

In the "Hallway" game, our agent again won all 15 of its matches. In this case, the results are even more significant because the prior probability of winning by chance is only about 20%. Roughly 60% of random games end in a tie.

Finally, in "Farmers", our agent performed better than the prior expectation of 35.15%, but did not win enough games for the results to be statistically significant. The agent needed to win 14 of its 25 games for us to be at least 95% confident that its success was not by chance. The automatically generated heuristics were quite successful on the first two games, and on the third game, did not hurt performance.

Related Work

Prior research on the feature discovery problem includes Fawcett's (1993) thesis work. Although Fawcett does not construct of a completely general game player, the feature discovery algorithm is applied in a game-playing setting. Features for the game of Othello are generated from a game description with syntax somewhat like GDL. Features are discovered by applying transformation operators on existing features, beginning with the goal condition itself, in a kind of search through feature space. It is unclear how dependent the method's success is on the STRIPS-style domain theory, but it may be possible to apply the same technique in GGP.

The most relevant work to general game playing is Barney Pell's Metagamer (Pell 1993). This work addresses the space of chess-like games, broad enough to include Checkers, Chinese Chess, Go and many more variants without common names. Again, because the domain representation was constructed as part of the work, it is not obvious that the techniques could directly apply in the GGP setting. This work also addresses the interesting problem of automatically generating novel games from a distribution of possibilities

Finally, there is an interesting, commercially available general game playing system called Zillions of Games [2]. Games are described using a higher-level language than GDL. The system comes with several games and an agent opponent to play against. As far as we are aware, though, this agent is not able to perform feature discovery or heuristic construction in a completely general way on its own.

Conclusion

We have introduced algorithms for discovering features in a formal game description and generating heuristic evaluation functions from them. These methods are integrated with theorem proving and heuristic search algorithms into a complete agent for general game playing. By building on these techniques, we can continue to make progress toward the ongoing general game playing challenge.

References

Campbell, M.; Jr., A. J. H.; and Hsu, F. H. 2002. Deep blue. *Artificial Intelligence* 134(1–2):57–83.

Fawcett, T. E. 1993. Feature discovery for problem solving systems, PhD thesis, University of Massachusetts, Amherst.

Genesereth, M., and Love, N. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2).

Genesereth, M. 1991. Knowledge interchange format. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Second Intl. Conference (KR'91)*.

Knuth, D. E., and Moore, R. W. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence* 6(4):293–326.

Korf, R. E. 1985. Iterative deepening: An optimal admissable tree search. *Artificial Intelligence* 27:97–109.

Pell, B. 1993. Strategy generation and evaluation for meta-game playing. PhD thesis, University of Cambridge.

Schaeffer, J.; Culberson, J. C.; Treloar, N.; Knight, B.; Lu, P.; and Szafron, D. 1992. A world championship caliber checkers program. *Artificial Intelligence* 53(2-3):273–289.

Schaeffer, J. 1989. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11:1203–1212.

Sturtevant, N. R., and Korf, R. E. 2000. On pruning techniques for multi-player games. In *Procs. AAAI-00*, 201–207.

Tesauro, G. 1994. Td-gammon, a self-teaching backgammon program, achieves masterlevel play. *Neural Computation* 6:215–219.

[2] `http://www.zillions-of-games.com/`