Fluxplayer: A Successful General Game Player

Stephan Schiffel and **Michael Thielscher**
Department of Computer Science
Dresden University of Technology
{stephan.schiffel,mit}@inf.tu-dresden.de

Abstract

General Game Playing (GGP) is the art of designing programs that are capable of playing previously unknown games of a wide variety by being told nothing but the rules of the game. This is in contrast to traditional computer game players like Deep Blue, which are designed for a particular game and can't adapt automatically to modifications of the rules, let alone play completely different games. General Game Playing is intended to foster the development of integrated cognitive information processing technology. In this article we present an approach to General Game Playing using a novel way of automatically constructing a position evaluation function from a formal game description. Our system is being tested with a wide range of different games. Most notably, it is the winner of the AAAI GGP Competition 2006.

Introduction

General Game Playing is concerned with the development of systems that can play well an arbitrary game solely by being given the rules of the game. This raises a number of issues different from traditional research in game playing, where it is assumed that the rules of a game are known to the programmer. Writing a player for a particular game allows to focus on the design of elaborate, game-specific evaluation functions (e.g., (Morris 1997)) and libraries (e.g, (Schaeffer *et al.* 2005)). But these game computers can't adapt automatically to modifications of the rules, let alone play completely different games.

Systems able to play arbitrary, unknown games can't be given game-specific knowledge. They rather need to be endowed with high-level cognitive abilities such as general strategic thinking and abstract reasoning. This makes General Game Playing a good example of a challenge problem which encompasses a variety of AI research areas including knowledge representation and reasoning, heuristic search, planning, and learning. In this way, General Game Playing also revives some of the hopes that were initially raised for game playing computers as a key to human-level AI (Shannon 1950).

In this paper, we present an approach to General Game Playing which combines reasoning about actions with heuristic search. Our focus is on techniques for constructing search heuristics by the automated analysis of game specifications. More specifically, we describe the following functionalities of a complete General Game Player:

1. Determining legal moves and their effects from a formal game description requires reasoning about actions. We use the Fluent Calculus and its Prolog-based implementation FLUX (Thielscher 2005).
2. To search a game tree, we use non-uniform depth-first search with iterative deepening and general pruning techniques.
3. Games which cannot be fully searched require the automatic construction of evaluation functions from formal game specifications. We give the details of a method that uses Fuzzy Logic to determine the degree to which a position satisfies the logical description of a winning position.
4. Strategic, goal-oriented play requires to automatically derive game-specific knowledge from the game rules. We present an approach to recognizing structures in game descriptions.

All of these techniques are independent of a particular language for defining games. However, for the examples given in this paper we use the Game Description Language developed by Michael Genesereth and his Stanford Logic Group; the full specification of syntax and semantics can be found at `games.stanford.edu`. We also refer to a number of different games whose formal rules are available on this website, too. Since 2005 the Stanford Logic Group holds an annual competition for General Game Players. The approach described in this paper has been implemented in the system Fluxplayer, which has won the second AAAI General Game Playing competition.

Theorem Proving/Reasoning

Games can be formally described by an axiomatization of their rules. A symbolic representation of the *positions* and *moves* of a game is needed. For the purpose of a modular and compact encoding, positions need to be composed of *features* like, for example, `(cell ?x ?y ?p)` representing that `?p` is the contents of square `(?x,?y)` on a chessboard.[1] The moves are represented by symbols, too,

[1]Throughout the paper, we use KIF (the *Knowledge Interchange Format*), where variables are indicated by a leading "?".

like (move ?u ?v ?x ?y) denoting to move the piece on square (?u,?v) to (?x,?y).

In the following, we give a brief overview of the Game Description Language. This language is suitable for describing finite and deterministic n-player games ($n \geq 1$) with complete information. GDL is purely axiomatic, so that no prior knowledge (e.g., of geometry or arithmetics) is assumed. The language is based on a small set of keywords, that is, symbols which have a predefined meaning. A general game description consists of the following elements.

- The **players** are described using the keyword (role ?p), e.g., (role white).
- The **initial position** is axiomatized using the keyword (init ?f), for example, (init (cell a 1 white_rook)).
- **Legal moves** are axiomatized with the help of the keywords (legal ?p ?m) and (true ?f), where the latter is used to describe features of the current position. An example is given by the following implication:

```
(<= (legal white (move ?x ?v ?x ?y))
    (true (cell ?x ?v white_pawn))
    (true (cell ?x ?y blank))
    ...)
```

- **Position updates** are axiomatized by a set of formulas which entail all features that hold after a move, relative to the current position and the moves taken by the players. The axioms use the keywords (next ?f) and (does ?p ?m), e.g.,

```
(<= (next (cell ?x ?y ?p))
    (does ?player (move ?u ?v ?x ?y))
    (true (cell ?u ?v ?p)))
```

- The **end of a game** is axiomatized using the keyword terminal, for example,

```
(<= terminal checkmate)
(<= terminal stalemate)
```

  where checkmate and stalemate are auxiliary, domain-dependent predicates which are defined in terms of the current position, that is, with the help of predicate true.
- Finally, the **result** of a game is described using the keyword (goal ?p ?v), e.g.,

```
(<= (goal white 100)
    checkmate (true (control black)))
(<= (goal black 0)
    checkmate (true (control black)))
(<= (goal white 50) stalemate)
(<= (goal black 50) stalemate)
```

  where the domain-dependent feature (control ?p) means that it's player's ?p turn.

In order to be able to derive legal moves, to simulate game play, and to determine the end of a game, a general game playing system needs an automated theorem prover. The first thing our player does when it receives a game description is to translate the GDL representation to Prolog. This allows for efficient theorem proving. More specifically, we use the Prolog based Flux system (Thielscher 2005) for reasoning. Flux is a system for programming reasoning agents and for reasoning about actions in dynamic domains. It is based on the Fluent Calculus and uses an explicit state representation and so called state-update axioms for calculating the successor state after executing an action. Positions correspond to states in the Fluent Calculus and features of a game, i.e. atomic parts of a position, are described by fluents.

Search Algorithm

The search method used by our player is a modified iterative-deepening depth-first search with two well-known enhancements: transposition tables (Schaeffer 1989) for caching the value of states already visited during the search; and history heuristics (Schaeffer 1989) for reordering the moves according to the values found in lower-depth searches. For higher depths we apply non-uniform depth-first search. With this method the depth limit of the iterative deepening search is only used for the move that got the highest value in a lower-depth search and is gradually lowered for the rest of the moves until a lower bound is reached. For the entry in the transposition table the maximal search depth of the best move in a state is used. This method allows for higher maximal depth of the search, especially with big branching factors. In practice, this increases the probability that the search reaches terminal states earlier in the match because all games considered end after a finite number of steps. Thus, non-uniform depth-first search helps to reduce the horizon problem in games ending after a limited number of steps. In those games the heuristic evaluation function might return a good value although the goal is in fact not reachable anymore because there are not enough steps left. A deeper search helps to avoid bad terminal states in that case. There is another, probably more important rationale behind non-uniform depth-limited search: It helps filling the transposition table with states that will be looked up again and therefore speeds up search in the future. If we search the state space following a move we do not take, it is unlikely that we will encounter these states again in the next steps of the match.

Depending on the type of the game (single-player vs. multi-player, zero-sum game vs. non-zero-sum game) we use additional pruning techniques e.g., alpha-beta-pruning for two-player games. Our player also decides between turn-taking and simultaneous-moves games. For turn-taking games the node is always treated as a maximization node for the player making the move. So we assume each player wants to maximize his own reward. For simultaneous-moves games we make a paranoid assumption: we serialize the moves of the players and move first. Thereby we assume that all our opponents know our move. In effect of this we can easily apply pruning techniques to the search but the assumption may lead to suboptimal play. There is currently work in progress to use a more game-theoretic approach including the calculation of mixed strategy profiles and Nash-equilibria.

Heuristic Function

Because in most games the state space is too big to be searched exhaustively, it is necessary to bound the search depth and use a heuristic evaluation function for non-terminal states. A general game playing system in our setting must be able to play all games that can be described with the GDL. Thus, it is not possible to provide a heuristic function beforehand that depends on features specific for the concrete game at hand. Therefore the heuristics function has to be generated automatically at runtime by using the rules given for the game.

The idea for our heuristic evaluation function is to calculate the degree of truth of the formulas defining predicates `goal` and `terminal` in the state to evaluate. The values for `goal` and `terminal` are combined in such a way that terminal states are avoided as long as the goal is not fulfilled, that is, the value of `terminal` has a negative impact on the evaluation of the state if goal has a low value and a positive impact otherwise.

The main idea is to use fuzzy logic, that is, to assign the values 0 or 1 to atoms depending on their truth value and use some standard t-norm and t-co-norm to calculate the degree of truth of complex formulas. However this approach has undesirable consequences. Consider the following evaluation function for GDL formulas in a game state z and let $p := 1$:

$$\begin{aligned}
eval(a,z) &= \begin{cases} p, \text{if } a \text{ holds in the state } z \\ 1-p, \text{otherwise} \end{cases} \\
eval(f \wedge g, z) &= T(eval(f,z), eval(g,z)) \\
eval(f \vee g, z) &= S(eval(f,z), eval(g,z)) \\
eval(\neg f, z) &= 1 - eval(f,z)
\end{aligned}$$

where a denotes a GDL atom, f, g are GDL formulas, and T denotes an arbitrary t-norm with associated t-co-norm S.

Now consider a simple blocks world domain with three blocks a, b and c and the goal $g = on(a,b) \wedge on(b,c) \wedge ontable(c)$. In this case we would want $eval(g,z)$ to reflect the number of subgoals solved. However, as long as one of the atoms of the goal is not fulfilled in z, $eval(g,z) = 0$.

To overcome this problem we use values different from 0 or 1 for atoms that are false or true respectively, that means $0.5 < p < 1$ in the definition of $eval$ above. This solves the problem of the blocks world domain described above, as long as we use a continuous t-norm T, that is, which satisfies the condition $x_1 < x_2 \wedge y > 0 \supset T(x_1,y) < T(x_2,y)$. One example for such a t-norm would be $T(x,y) = x * y$. However this solution introduces a new problem. Because of the monotonicity of T, $eval(a_1 \wedge \ldots \wedge a_n, z) \to 0$ for $n \to \infty$. Put in words, the evaluation says the formula $a_1 \wedge \ldots \wedge a_n$ is false (value is near zero) for large n even if all a_i hold in z.

To overcome this problem, we use a threshold t with $0.5 < t \leq p$, with the following intention: values above t denote true and values below $1-t$ denote false. The truth function we use for conjunction is now defined as:

$$T(a,b) = \begin{cases} max(T'(a,b), t), & \text{if } min(a,b) > 0.5 \\ T'(a,b) & \text{otherwise} \end{cases}$$

where T' denotes an arbitrary standard t-norm. This function together with the associated truth function for disjunctions ($S(a,b) = 1 - T(1-a, 1-b)$) ensures that formulas that are true always get a value greater or equal t and formulas that are false get a value smaller or equal $1-t$. Thus the values of different formulas stay comparable. This is necessary, for example, if there are multiple goals in the game. The disadvantage is that T is not associative, at least in cases of continuous t-norms T', and is therefore not a t-norm itself in general. The effect of this is that the evaluation of semantically equivalent but syntactically different formulas can be different. However by choosing an appropriate t-norm T' it is possible to minimize that effect.

For the t-norm T' we use an instance of the Yager family of t-norms:

$$\begin{aligned}
T'(a,b) &= 1 - S'(1-a, 1-b) \\
S'(a,b) &= (a^q + b^q)^{\frac{1}{q}}
\end{aligned}$$

The Yager family captures a wide range of different t-norms. Ideally we would want a heuristic and thus a pair of t-norm and t-co-norm that is able to differentiate between all states that are different with respect to the goal and terminal formulas. In principal this is possible with the Yager family of t-norms. By varying q one can choose an appropriate t-norm for each game, depending on the structure of the goal and terminal formulas to be evaluated, such that as many states as possible that are different with respect to the goal and terminal formulas are assigned a different value by the heuristic function. However, at the moment we just use a fixed value for q which seems to work well with most games currently available on `http://games.stanford.edu`.

According to the definitions above, the evaluation function has the following property, which shows its connection to the logical formula:

$$\begin{aligned}
(\forall f, z)\, eval(f,z) \geq t > 0.5 &\iff holds(f,z) \\
(\forall f, z)\, eval(f,z) \leq 1 - t < 0.5 &\iff \neg holds(f,z)
\end{aligned}$$

where $holds(f,z)$ denotes that formula f holds in state z.

The heuristic evaluation function for a state z in a particular game is defined as follows:

$$\begin{aligned}
h(z) &= \frac{1}{\sum_{gv \in GV} gv} * \bigoplus_{gv \in GV} h(gv,z) * gv/100 \\
h(gv,z) &= \begin{cases} eval(goal(gv) \vee term, z), \text{if } goal(gv) \\ eval(goal(gv) \wedge \neg term, z), \text{if } \neg goal(gv) \end{cases}
\end{aligned}$$

$\bigoplus$ denotes a product t-co-norm sum, GV is the domain of goal values, $goal(gv)$ is the (unrolled) goal formula for the goal value gv and $term$ is the (unrolled) terminal formula of the game. That means the heuristics of a state is calculated by combining heuristics $h(gv,z)$ for each goal value gv of the domain of goal values GV weighted by the goal value gv with a product t-co-norm (denoted by $\bigoplus$). The heuristics for each possible goal value is calculated as the evaluation of the disjunction of the goal and terminal formulas in case the goal is fulfilled, that is, the heuristics tries to reach a terminal state if the goal is reached. On the other hand the heuristics tries to avoid terminal states as long as the goal is not reached.

Identifying Structures

The evaluation function described above can be further improved by using the whole range of values between 0 and 1 for atoms instead of assigning fixed values $1-p$ for false and p for true atoms. The idea is to detect structures in the game description which can be used for non-binary evaluations like successor relations, order relations, quantities or game boards. The approach is similar to the one of (Kuhlmann, Dresner, & Stone 2006) but with some improvements.

Unlike (Kuhlmann, Dresner, & Stone 2006), who use the syntactical structure of the rules, we exploit semantical properties of the predicates to detect static structures, which are independent of the state of the game. For example, binary relations that are antisymmetric, functional and injective are considered as successor relations. Because we use semantical instead of syntactical properties, we can also detect higher-level predicates like order relations, that is, binary relations which are transitive and antisymmetric. This is difficult to do when relying on the syntax of the rules, because there are many semantically equivalent but syntactically different descriptions of a predicate. The properties of the predicates can be proved quite easily because all domains are finite.

Dynamic structures like game boards and quantities are detected in the same way as described in (Kuhlmann, Dresner, & Stone 2006). However our definition of a game board is broader in that we allow arbitrary dimensions for boards. We can also deal with boards where only some of the coordinates, i.e. input arguments, are ordered, in which case the fluent possibly describes multiple ordered boards. An example for this case is the fluent `(cell ?b ?y ?x ?c)` in the Racetrack game (the final game in the AAAI GGP Competition 2005), a two-player racing game where each player moves on his own board. The arguments `?y` and `?x` are the coordinates of the cell on board `?b` and `?c` is the content of the cell. Only the domains of `?x` and `?y` are ordered, but the input arguments are `?b`, `?x` and `?y`.

For deciding if a fluent describes a board or a quantity we need to know the fluent's input and output arguments as well as the information if a certain argument of a fluent is ordered. We compute input and output arguments of all fluents in the same way as (Kuhlmann, Dresner, & Stone 2006).

For the decision if an argument of a fluent is ordered it is necessary to know the domains of the arguments of fluents and if there is a successor relation for this domain. The domains, or rather supersets of the domains, of all predicates and functions of the game description are computed by generating a dependency graph from the rules of the game description. The nodes of the graph are the arguments of functions and predicates in game description, and there is an edge between two nodes whenever there is a variable in a rule of the game description that occurs in both arguments. Connected components in the graph share a (super-)domain.

Figure 1 shows the dependency graph for the following game rules describing an ordinary step counter:

```
(succ 0 1) (succ 1 2) (succ 2 3)
(init (step 0))
(<= (next (step ?x))
    (true (step ?y)) (succ ?y ?x))
```

The arguments of the function `step` and the predicate `succ` are all connected in the graph and thus share the same domain. The computed set of constants $\{0, 1, 2, 3\}$ is actually a superset of the real domain of the arguments of `succ`. For example, `3` can't occur in the first argument of `succ`. We disregard this fact because we are more interested in the dependencies between different functions and predicates than in the actual domains.

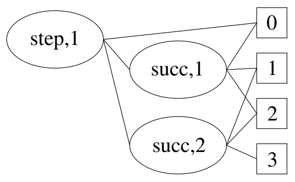

Figure 1: A dependency graph for calculating domains of functions and predicates. (Ellipses denote arguments of fluents or predicates and squares denote constants.)

Using identified structures for the heuristics

The heuristic evaluation function is improved by introducing non-binary evaluations of the atoms that correspond to identified structures.

The evaluation of an order relation r is computed as

$$eval(r(a,b),z) = \begin{cases} t + (1-t) * \frac{\Delta(a,b)}{|dom(r)|}, \text{if } r(a,b) \\ (1-t) * (1 - \frac{\Delta(b,a)}{|dom(r)|}), \text{if } \neg r(a,b) \end{cases}$$

where $\Delta(a,b)$ is the number of steps needed for reaching b from a with the successor function that is the basis of this order relation, and $|dom(r)|$ denotes the size of the domain of the arguments of r. This evaluation function has advantages over a binary evaluation which just reflects the truth value of the order relation: It prefers states with a narrow miss of the goal over states with a strong miss.

The evaluation of atoms of the form `(true f)` can be non-binary if the `f` in question describes an ordered game board or a quantity. For ordered game boards this evaluation reflects the distance (computed with a normalized city-block metrics) of the position of a piece on the board in the current state to the goal position. If there are multiple matching pieces in a state like in Chinese Checkers where the pieces of each player are indistinguishable, the evaluation is the mean value of the distances for all matching pieces.

For example, the evaluation function for a two-dimensional ordered board with the coordinates x and y is:

$$eval(f(x,y,c),z) = \frac{1}{N} * \sum_{f(x',y',c) \in z} \frac{1}{2} * \left(\frac{\Delta(x,x')}{|dom(f,1)|} + \frac{\Delta(y,y')}{|dom(f,2)|} \right)$$

where N is the number of occurrences of some $f(x', y', c)$ in z, $|dom(f,i)|$ denotes the size of the domain of the i-th argument of f, and $\Delta(x,x')$ is the number of steps between x and x' according to the successor function that induces the order for the domain. The function can easily be extended to boards of arbitrary dimensio.

If the fluent f of the atom `(true f)` is a quantity (or a board where the cell's state is a quantity), the evaluation is based on the difference between the quantity in the current state and the quantity in the atom to evaluate. E.g., the evaluation function for a unary quantity fluent f like the step counter is:

$$eval(f(x), z) = \frac{\Delta(x, x')}{|dom(f, 1)|}, \text{if } f(x') \in z$$

This function can be extended easily to an arbitrary (non-zero) number of input and output arguments.

Evaluation

It is difficult to find a fair evaluation scheme for general game playing systems because the performance of a general game player might depend on many different factors, e.g., the game played, the actual description of the game, the opponents and the time given for analysis of the game as well as for actual game play. For general game playing, there exists no predefined set of benchmark problems to compare your players with, as opposed to e.g., SAT solvers or planners. So instead of developing our own set of benchmark problems and running a lot of experiments with different parameters against random opponents, we choose to use the results of more than 400 matches (more than 70 for each player) of the AAAI GGP Competition 2006 for determining which parts of our evaluation function affected good play for which games. The advantage of using this data is that we were playing against real opponents and thus we can directly compare the performance of our player to that of other state of the art systems.

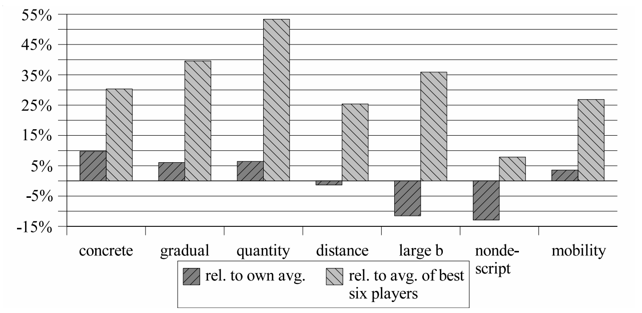

Figure 2: The chart shows the average score of Fluxplayer for certain games relative to (1) the overall average score of Fluxplayer and (2) the average score of the best six players for the games with the respective property.

Figure 2 depicts the performance of Fluxplayer separately for games with the following seven properties:

- a *concrete* goal state, i.e., the goal is one conjunction of ground fluents not necessarily describing a complete goal state;
- *gradual* goal values, i.e., goal values < 100 for reaching only a part of the goal;
- the goal involves a *quantity*, i.e., to reach a certain amount of something;
- the goal involves a position of a piece on an ordered board such that a *distance* measure applies;
- a large branching factor b, i.e. $b \geq 20$, for some state that occurred during a match;
- a *nondescript* goal, i.e., all fluents that occur in the goal have the same truth value in all states of the same search depth;
- the goal or terminal condition amounts to a *mobility* heuristics, which is the case, e.g., for games ending if there are no more legal moves.

Of course these properties do not define distinct categories, that is, there are games that have more than one of the properties. Still most games have only few of these properties and none of the sets of games is a subset of another. Thus, some statements can be made.

We see that Fluxplayer performs better than its own average for games with concrete goals and gradual goals. If a game has a concrete goal then, because of the evaluation function, those non-terminal states are preferred in which more of the fluents of the conjunction hold. The evaluation function also takes advantage of gradual goals. Thus, the good performance in games with concrete goals and gradual goals indicates that our particular construction of the evaluation function has a fair share in the success. The same argument holds for games with goals involving quantities, indicating that our detection of quantities and order relations has a positive influence on the performance. For games involving quantities we can also see that Fluxplayer performs much better than all the opponents, which probably means that the other players do not take as much advantage of this information. We can see that Fluxplayer got average scores for games with distance measures and that the winning margin of Fluxplayer for those games is smaller. For games with a high branching factor the scores of Fluxplayer are, not surprisingly, below its average. After all the search depth reached in those games is much smaller. Still Fluxplayer seems to handle those games much better than its opponents, which, besides games with other advantageous properties, might be caused by the higher maximal depth reached during search because of the non-uniform search.

The main disadvantage of our approach is the dependency on the goal description, as can be seen by the poor performance for games with nondescript goals. This is already alleviated by the fact that many of these games end if there are no more moves left, in which case our evaluation function for the terminal condition automatically amounts to a mobility heuristics. Games with nondescript goals are an important area for improvements of our evaluation function.

Discussion

The evaluation function obtained by the method described above is directly based on the goal description of the game. The generation of the heuristics is much faster than learning-based approaches. Thus our approach is particularly well-suited for games with a big state space and sparse goal states. In both cases, with learning-based approaches one has to play many random matches to have significant data. Based on the same rationale our approach has advantages for games with little time for learning parameters of the heuristic function.

However, depending on the game description our evaluation function can be more complex to compute than learned features, or it might not be able to differentiate between non-terminal states at all. Decomposition and abstraction techniques like they are described in (Fawcett & Utgoff 1992) might be used to overcome the first problem. We want to tackle the second problem by formal analysis of the rules of the game for discovering relations between properties of different states and thereby detecting features that are important for reaching the goal.

Another advantage of our approach regarding possible future extensions of the general game playing domain is the direct applicablility to domains with incomplete information about the current position. Flux, which is used for reasoning, as well as the heuristic evaluation function are in principle able to handle incomplete-information games.

Related Work

A work in the same setting is (Kuhlmann, Dresner, & Stone 2006). The authors consider the same problem but use a different approach for heuristics construction. A set of candidate heuristics is constructed, which are based on features detected in the game, but the goal description is not taken into account. The process of selecting which of the candidate heuristics is leading to the goal remains an open challenge.

Previous work on general game playing includes Barney Pell's Metagamer (Pell 1993) which addresses the domain of Chess-like board games. Fawcett (Fawcett 1990) applies a feature discovery algorithm to a game of Othello. Features are generated by inductively applying transformation operators starting with the goal description. Discovered features need to be trained. Much research has been done on heuristic-search planning (Bonet & Geffner 2001), which might be reused at least for single-player games.

However, most of the work depends on STRIPS-style domain descriptions. Further investigations are necessary in order to determine which of the techniques can be applied to domain descriptions in GDL, and which can be adapted for multi-player games.

Conclusion

We have presented an approach to General Game Playing which combines reasoning about actions with heuristic search. The main contribution is a novel way of constructing a heuristic evaluation function by the automated analysis of game specifications. Our search heuristics evaluates states with regard to the incomplete description of goal and terminal states given in the rules of the game. It takes advantage of features detected in the game like boards, quantities and order relations.

Our General Game player performed well in a wide variety of games, including puzzles, board games, and economic games. However there is ample room for improvement: We are currently working on enhancing the translation of the game description to Prolog in order to make reasoning and especially the calculation of successor states more efficient and therefore the search faster. There is also work in progress directed at an improved evaluation of non-leaf nodes of the search tree, which uses methods from game theory and allows better opponent modeling.

A number of problems of the heuristics need to be addressed. This includes a more efficient evaluation of complex goal descriptions, which might be achieved by using abstraction and generalization techniques to focus on important features and disregard features which are less important but expensive to compute.

We plan to use formal analysis of the game description to get a more thorough picture of the function of each part of the game description and their connections. This information can then be used to increase the efficiency of the theorem prover as well as detecting new features of the game, which might improve the heuristics.

For directing future research we need to analyze the impact of the different evaluation functions on the game playing performance. For this analysis it is necessary to define realistic opponent models and problem sets. Ideally, we would want to have a range of benchmark domains and opponents such that objective comparisons between different general game playing systems are possible.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Fawcett, T. E., and Utgoff, P. E. 1992. Automatic feature generation for problem solving systems. In *Proc. of ICML*, 144–153. Morgan Kaufmann.

Fawcett, T. E. 1990. Feature discovery for inductive concept learning. Technical Report UM-CS-1990-015, Department of Computer Science, University of Massachusetts.

Kuhlmann, G.; Dresner, K.; and Stone, P. 2006. Automatic heuristic construction in a complete general game player. In *Proc. of AAAI*, 1457–62.

Morris, R., ed. 1997. *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*. AAAI Press.

Pell, B. 1993. Strategy generation and evaluation for metagame playing.

Schaeffer, J.; Björnsson, Y.; Burch, N.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2005. Solving checkers. In *Proc. of IJCAI*, 292–297.

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

Shannon, C. 1950. Programming a computer for playing chess. *Philosophical Magazine 7* 41(314):256–275.

Thielscher, M. 2005. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Applied Logic Series. Kluwer.