Heuristic Interpretation of Predicate Logic Expressions in General Game Playing

Daniel Michulke
Department of Computer Science
Dresden University of Technology
daniel.michulke@mailbox.tu-dresden.de

Abstract

Unlike traditional game playing, General Game Playing (GGP) is concerned with agents capable of playing classes of games. Given the rules of an unknown game, the agent is supposed to play well without human intervention. For this purpose, agent systems using game tree search need to automatically construct a state value function to guide search.

This state value function is often either probabilistic as in Monte-Carlo Systems and thus most likely unable to compete with deterministic functions in games like chess; or it is expensive in construction due to feature generation and learning-based selection and weighting mechanisms.

In this work we present an alternative method that derives features from the goal conditions stated in the game rules, avoiding thereby the disadvantages of other approaches. The paper structures and generalizes some known approaches, shows new ways of deriving features and demonstrates how to use these features as part of an evaluation function. Experiments demonstrate both a high effectiveness and generality.

1 Introduction

In General Game Playing, agents are supposed to play games they have never seen before. In a typical competition setup the agent receives the rules of a game and has a few minutes time until the corresponding match starts. Since the games are arbitrary deterministic games with complete information (such as Chess, Go, Tic-Tac-Toe, Rock-Paper-Scissors), the agent cannot rely on any preprogrammed behavior. Instead, it has to come up with a strategy that fits the game.

An important aspect of this strategy is the state evaluation function that evaluates states and is used to guide the agent to states with a high value. There are two basic types of evaluation functions:

Monte-Carlo based functions as used in (Finnsson and Björnsson 2008) evaluate a state by performing playouts, that is, playing random moves until a terminal state is reached. After several of such playouts, the average of the different match outcomes is considered the state value. The problem is that this evaluation assumes a random opponent behavior and stands no chance against informed and therefore non-random opponents in games with well-studied evaluation functions such as Chess. Extensions to the approach such as (Finnsson and Björnsson 2010) moderate this effect, but do not solve the problem.

The alternative is a deterministic evaluation based on the state itself. To obtain such an evaluation function, agents such as (Kuhlmann, Dresner, and Stone 2006; Clune 2007) derive candidate features from expressions in the game description, evaluate them to obtain a measure of usefulness for the game and finally put the features together, possibly weighted, to form an evaluation function. While the approach is perfectly reasonable, the evaluation of candidate features is usually time-consuming which is especially critical in a competition setup.

Another type of deterministic evaluation avoids this problem by using a function that is essentially a fuzzified version of the goal condition as stated in the game rules (Schiffel and Thielscher 2007; Michulke and Thielscher 2009). After transforming the goal condition to a propositional fuzzy evaluation function, it is improved by substituting specific expressions by more expressive features. In contrast to the feature detection and weighting approach, the features applied have a direct justification since they occur in the goal condition. Also, they need no weights assigned as the fuzzy goal condition provides the context for dealing with the feature output, i.e. weights them implicitly.

In this work we will present a generalized feature detection mechanism that builds on the latter two approaches. We show how to detect and apply features for the use in such an evaluation function. In contrast to other works published until now, we focus on a clear algorithmic description of 1. the construction process of the evaluation function and 2. the use of the evaluation function. Note that in this paper we do not intend to present a competitive GGP agent since the topic of straightforward feature detection and weighting is complex enough. Instead, we aim to present a general technique of deriving an evaluation function from a given domain theory. We believe the techniques presented should be part of the repertoire of any non-probabilistic agent since the resulting evaluation function represents a sound starting point for state evaluation without the disadvantages related to standard approaches for detecting, weighting and combining features.

The features discussed will cover the most important features used in the cited sources, introduce some features not yet used and, most importantly, show how to detect, confirm and apply these features without a costly feature evaluation and weighting phase. An evaluation will prove both generality and utility of the features.

The rest of this work is structured as follows: In the remainder of this section we will briefly introduce GDL and show in the next section how the goal condition of the game can be used to construct a simple evaluation function using (propositional) t-norm fuzzy logics and ground-instantiation. We proceed by introducing features in section 3 that we distinguish as expression features and fluent features. We evaluate these features in section 4 and summarize our results in section 5.

Game Description Language The language used for describing the rules of games in the area of General Game Playing is the Game Description Language(Love et al. 2008) (GDL). The GDL is an extension of Datalog with functions, equality, some syntactical restrictions to preserve finiteness, and some predefined keywords.

The following is a partial encoding of a Tic-Tac-Toe game in GDL. In this paper we use Prolog syntax where words starting with uppercase letters stand for variables and the remaining words are constants.

```
1  role(xplayer). role(oplayer).
2  init(cell(1,1,b)). init(cell(1,2,b)).
3  init(cell(2,3,b)). ... init(cell(3,3,b)).
4  init(control(xplayer)).
5  legal(P, mark(X, Y)) :-
6    true(control(P)), true(cell(X, Y, b)).
7  legal(P,noop) :- role(P),not true(control(P)).
8  next(cell(X,Y,x)) :- does(xplayer,mark(X,Y)).
9  next(cell(X,Y,o)) :- does(oplayer,mark(X,Y)).
10 next(cell(X,Y,C)) :-
11   true(cell(X,Y,C)), C \= b.
12 next(cell(X,Y,b)) :- true(cell(X,Y,b)),
13   does(P, mark(M, N)), (X \= M ; Y \= N).
14 goal(xplayer, 100) :- line(x).
15 ...
16 terminal :- line(x) ; line(o) ; not open.
17 line(P) :- true(cell(X, 1, P)),
18   true(cell(X, 2, P)), true(cell(X, 3, P)).
19 ...
20 open :- true(cell(X, Y, b)).
```

The first line declares the roles of the game. The unary predicate `init` defines the properties that are true in the initial state. Lines 5-7 define the legal moves of the game, e.g., `mark(X,Y)` is a legal move for role `P` if `control(P)` is true in the current state (i.e., it's `P`'s turn) and the cell `X,Y` is blank (`cell(X,Y,b)`). The rules for predicate `next` define the properties that hold in the successor state, e.g., `cell(M,N,x)` holds if `xplayer` marked the cell `M,N` and `cell(M,N,b)` does not change if some cell different from `M,N` was marked. Lines 14 to 16 define the rewards of the players and the condition for terminal states. The rules for both contain auxiliary predicates `line(P)` and `open` which encode the concept of a line-of-three and the existence of a blank cell, respectively.

We will refer to the arguments of the GDL keywords **`init`**, **`true`** and **`next`** as fluents. In the above example, there are two different types of fluents, `control(X)` with `X` $\in$ `{xplayer, oplayer}` and `cell(X, Y, C)` with `X`, `Y` $\in$ `{1, 2, 3}` and `C` $\in$ `{b, x, o}`. Any subset of the set of these fluents constitutes a state s. By applying the **`next`** rules on the current state s and the set of terms **`does`**`(P, A)` for each role `P` and its selected action `A`, one can derive the successor state s' of s.

2 The Evaluation Function

The evaluation function takes as argument a state and returns a value that indicates the degree of preference of the state. The function is used to find the best among different actions by deriving the successor state(s) induced by that action, evaluating them and selecting the action that leads to the best-valued successor state(s).

Since in General Game Playing each game is different, there is no generally useful evaluation function and an agent has to construct a function automatically depending on the game. One possibility is to take the game's goal function that maps states to a goal value. However, since goal function in GDL is "crisp", it does not contain information to what extent a state fulfills the underlying goal condition. Moreover, it most often only distinguishes between terminal states. By fuzzifying the goal function both problems can be addressed.

Construction After receiving the rules we preprocess the goal conditions of the game to obtain an evaluation function. Since each goal condition describes an existentially quantified predicate logic expression, our algorithm operates on these expressions. The following expansion algorithm takes as first argument an existentially quantified expression and transforms it recursively into a tree-like structure that represent the evaluation function:

```
expand(true(Fluent), true(Fluent)) :- !.
expand(not(Expr), not(Child)) :- !,
  expand(Expr, Child).
expand(Expr, and(Children)) :-
  is_conjunction(Expr, Conjuncts), !,
  expand_children(Conjuncts, Children).
expand(Expr, or(Children)) :-
  is_disjunction(Expr, Disjuncts), !,
  expand_children(Disjuncts, Children).
expand(Expr, R) :-
  is_game_specific_pred(Expr, Expansions), !,
  (Expansions = [OneExpansion] ->
    expand(OneExpansion, R)
  ;
    R = or(Children)
    expand_children(Expansions, Children)
  ).
expand(Expr, reason(Expr)).
expand_children([], []).
expand_children([C|Children], [E|Expansions]) :-
  expand(C, E),
  expand_children(Children, Expansions).
```

The predicates `is_conjunction/2`, `is_disjunction/2` and `is_game_specific_pred/2` analyze whether the expression is what the predicate expects it to be and returns, if confirmed, as second argument a list of the constituting expressions.

So given `Expr=(a(b), c(X), d)` it would hold `is_conjunction(Expr, [a(b), c(X), d])` while `is_disjunction(Expr, Conjuncts)` would fail since `Expr` is not a disjunction.

Obviously, the algorithm needs a little more to work the intended way. For instance, the predicate `is_conjunction/2` must not split conjunctions where the conjuncts share variables. So an expression like `is_conjunction((a(X), b(X), c(Y)), Conjuncts)` should yield a list of two conjuncts, namely `(a(X), b(X))` and `c(Y)`. Consequently, the expression `(a(X), b(X))` is not seen as a conjunction by our algorithm, but rather as an expression that will be delegated to the reasoner. Also we must not expand recursive expressions in `is_game_specific_pred/2`.

In the case of Tic-Tac-Toe, we could e.g. pass the expression **`goal`**`(xplayer, 100)` as first argument to the `expand` algorithm and receive a structure that represents the fuzzified evaluation of whether a given state fulfills the winning condition for role `xplayer`. We will refer to such a structure as evaluation tree.

State Evaluation The following is an example how to use this evaluation tree to obtain a single value that represents a propositional fuzzy evaluation of the goal condition applied on the state. The evaluation tree is passed as first argument and the state we want to evaluate as second argument. As last argument we receive the fuzzy value. All predicates with the prefix `fz_` and the predicates **`true`**`/1` and `false/1` are defined along with the fuzzy logic in the next section.

```
eval(not(Child), S, V) :-
  eval(Child, S, V1), not(V1, V).
eval(true(Fluent), S, V) :-
  fluent_holds(Fluent, S),
  !, fz_true(V).
eval(true(_Fluent), _, V) :- fz_false(V).
eval(and([]), _, V) :- !, true(V).
eval(and([C|Children]), S, V) :-
  eval(C, S, V1),
  eval(and(Children), S, V2),
  fz_and(V1, V2, V).
eval(or([]), _, V) :- !, false(V).
eval(or([C|Children]), S, V) :-
  eval(C, S, V1),
  eval(or(Children), S, V2),
  fz_or(V1, V2, V).
eval(reason(Expr), S, V) :-
  expression_holds(Expr, S),
  !, fz_true(V).
eval(reason(_Expr), _, V) :- fz_false(V).
```

Here we evaluate all leafs of the evaluation function structure (fluents and expressions) to the value defined by `fz_true` if they hold in the current state and to the `fz_false` value otherwise. These values are passed recursively up the evaluation structure using the predicates `fz_and`, `fz_or` and **`not`** and the value at the root node represents the fuzzy evaluation of the goal condition on the state. The predicates **`true`** and `false` just represent the upper and lower bound of the interval of truth values and must be set in accordance with the fuzzy logic.

Underlying Fuzzy Logic A useful propositional fuzzy logic could be any t-norm fuzzy logic, e.g.:

```
fz_true(0.9). true(1).
fz_false(0.1). false(0).
not(X, Y) :- Y is 1 - X.
fz_and(X, Y, Z) :- Z is Y * X.
fz_or(X, Y, Z) :- Z is X + Y - X * Y.
```

Fluxplayer (Schiffel and Thielscher 2007) also operates on the interval $[0, 1]$ of truth values, but uses an instance of the Yager family of t-norms.

We believe that the class of applicable fuzzy logics can be extended to more general classes than just t-norm fuzzy logics. In fact, for the GGP setting any "approximately correct" logic can be used since we will discuss features that return incorrect values with respect to the used fuzzy semantic, yet these features contribute well to the evaluation quality of a state. For the remainder of this paper we assume the t-norm fuzzy logic as given above.

Given the expansion and evaluation algorithms together with an appropriate propositional fuzzy logic, we can evaluate the degree of truth of the goal condition on a particular state. Practically, we obtain a measure to compare two states by comparing their state values and we can assume that the state with the higher state value is closer to the goal than the other state. Note however, the evaluation function up to now builds merely upon the logical structure of the goal expression. Distances or quantitative evaluations are ignored.

Ground-Instantiation Since the domains of all variables occurring in a game description are finite, it is possible to ground-instantiate a game by substituting any expression containing a variable with the disjunction of the expressions where that variable is substituted by all terms in the domain of the variable. We use ground-instantiation to deal with expressions that cannot be expanded further. In this way, it is e.g. possible to break down variable-sharing conjunctions to a disjunction of standard conjunctions and thus increase the part of the original predicate logic expression that can be evaluated using fuzzy logic. Note, however, that the number of ground-instantiations of an expression may grow exponentially with the number of variables. It is therefore sometimes necessary to use a mixed approach, instantiating some variables while leaving others uninstantiated. Expressions uninstantiated for whatever reason are directly evaluated by the reasoner.

After ground-instantiating an expression we may encounter (so called static) predicates in the resulting ground expressions that do not depend on the current state of the game. We can immediately evaluate them to true or false and hence do not need to include them in the evaluation function.

Our basic principle for constructing the evaluation tree is to keep all expressions uninstantiated for as long as possible to avoid growing the size of the evaluation function.

3 Features

We will present several features and show how to detect them in a predicate logic expression. This will require us to modify the `expand` algorithm given above. Due to space restrictions we cannot give a formal description of the requirements an expression needs to fulfill in order to be identified as a feature. However, we will discuss them informally in the corresponding paragraph. For each feature discussed we will give an example on how to utilize it by presenting additional rules for the above `eval` algorithm.

The features themselves represent heuristics and there are always situations in which the features produce the opposite of what a perfect state evaluation function would return. However, this problem is a general problem of game playing agents. It is addressed by using only features that predominantly improve the evaluation quality and by using many features such that over- and underestimations of feature values neutralize themselves.

3.1 Expression Features

We start with expression features that are derived from expressions containing variables, namely fluents with variables, conjunctions of conjuncts that share variables, and predicates containing variables. For the detection of expression features we specialize the `expand` algorithm such that instead of further expansion via ground-instantiation or simple delegation to the reasoner, we interpret the expression directly as feature, and add this feature as leaf to the evaluation tree. This has two main effects: First, by avoiding ground-instantiation the number of nodes in the evaluation tree is smaller and the function therefore faster to evaluate. And second, the resulting feature is more expressive than its otherwise expanded alternative.

In other words, evaluation becomes faster and estimates better at the same time.

Solution Cardinality

```
open :- true(cell(X, Y, b)).
```

The basic element for querying the current state of the match is the **`true`** statement with a fluent as argument. Since we apply ground-instantiation as late as possible, we may encounter fluents with non-ground arguments. An example is the above rule from the game Tic-Tac-Toe. The `open/0` predicate is used to determine whether there are blank cells and whether the match can therefore continue.

We impose a first interpretation on the body of the rule by counting the number of solutions in the expression. So instead of evaluating the expression to true or false, we evaluate it to false if there are no solutions, but give it a higher value for the more solutions there are. Assuming that states are relatively stable and the set of true fluents does not change radically from one state to another, a higher solution cardinality indicates that the expression is more likely to be true in successor states of the state under inspection. Besides, the evaluation function is smaller than its ground-instantiated variant (one node instead of 9 in the case of Tic-Tac-Toe) and the evaluation is therefore also faster than a nine-fold query of each individual ground fluent `cell(1, 1, b)`,... `cell(3, 3, b)`.

To use the feature, we change the `eval(true(Fluent), S, V)` rule in our evaluation function such that it is equal to the value of a standard disjunction of the ground fluents.

For the `open/0` example, the evaluation would look like:

```
eval(true(cell(X, Y, b)), S, V) :-
  count_solutions(cell(X, Y, b), S, SolCnt),
  count_instances(cell(X, Y, b), Inst),
  fz_generalized_or(SolCnt, Inst, V).
```

`count_instances/2` returns the number of valid instances for first argument (in this case 9) and `fz_generalized_or/3` is a helper that calculates the fuzzy value of a disjunction that is `Inst-SolCnt` times false and `SolCnt` times true.

We apply this interpretation whenever we encounter fluents with variables or expressions that we would otherwise pass to the reasoner. This feature is the same as the one Clune proposed(Clune 2007) under the same name. In his approach, however, solution cardinality features were constructed by first considering all expressions as solution cardinality features, then evaluating specific feature properties (stability, correlation to goal values and computation cost) on a number of test states and finally removing all candidate features that do not exhibit the desired properties to a sufficient degree. In contrast, our approach is much faster since the set of candidate features is greatly reduced and the correlation to goal values is implicitly determined through the context the fuzzy logic provides.

Order Conjunctions with shared variables adhere to a number of patterns. One example is a goal expression in the game Checkers which states

```
goal(white, 100) :-
  true(piece_count(white, W)),
  true(piece_count(black, B)),
  greater(W, B).
```

The `piece_count` fluent keeps track of the number of pieces each role has and `greater` is the axiomatized variant of the greater-than relation. Thus white achieves 100 points if it has more pieces than black.

Since domains typically have no more than a few hundred elements, we can quickly prove that the relation expressed by the predicate `greater/2` is antisymmetric, functional, injective and transitive and thus conclude that it represents a total order, possibly with the exception of the reflexive case. Therefore, we can map all elements in the order to natural numbers. Assuming that the fluents `piece_count(white, _)` and `piece_count(black, _)` occur exactly once in each state, we can identify an expression of the above type as order expression. Generally spoken, we identify an expression as order feature if we encounter a conjunction with sharing variables where at least one fluent variable occurs in a static predicate that represents a total order. In addition we require the fluents in the conjunction to occur exactly once in any state.

We evaluate the above example the following way:

```
eval(order(goal(white, 100)), S, V) :-
  fluent_holds(piece_count(white, W), S),
  fluent_holds(piece_count(black, B), S),
  term_to_int(greater_2, W, WI),
  term_to_int(greater_2, B, BI),
  domain_size(greater_2, Size),
  V is 0.5 + (WI-BI)/(2*Size).
```

The advantage of this type of order heuristics is most importantly a finer granularity in comparisons of elements: The return value `V` is higher for higher differences between the terms `W` and `B` and lower if both are approximately equal. A ground-instantiated version of this evaluation would not be able to distinguish between differences of $+2$ or $+20$. Besides, we reduce the size of the evaluation function by avoiding ground-instantiation. In this case, since there are 13 possible values of the `piece_count` fluent (0 to 12 pieces) for each role, we express 169 ground evaluations by one heuristic order expression.

The only disadvantage is that we do not see an easy way to express this relation while sticking to the truth values required by our fuzzy logic, thus we lose correctness at this point.

A structurally different feature with the same expressiveness is also detected by (Clune 2007) who looks for comparative features if solution cardinalities are found. However, it is unclear if his agent is able to exploit orders on other quantitative statements. Our approach is similar to that of (Schiffel and Thielscher 2007).

Relative Distances I - Distance Between Fluents Another pattern for conjunctions with shared variables is that of the relative distance. If two fluents occur in a conjunction and share the same variables at the same argument positions then these are candidates for relative distances. An example is the winning rule for `inky` in the game Pac-Man ("pacman3p"):

```
goal(inky, 100) :-
  true(location(pacman, X, Y)),
  true(location(inky, X, Y))).
```

In order to confirm that this pattern can be interpreted as relative distance pattern, we have to make sure that it is possible to calculate distances on the argument positions of the fluent where the variables occur. We do this by identifying, how the arguments of the fluent evolve over time. If, for instance, the fluent `location/3` represents a metric board where Pac-Man can move north, then there is most probably a rule in the GDL description of the game such that

```
next(location(pacman, X, Z)) :-
  true(location(pacman, X, Y)),
  a_predicate(Y, Z).
```

By identifying static predicates such as `a_predicate/2`, one can derive a graph where all valid instantiations of `a_predicate/2` are the edges.

Most often, the relation represented by `a_predicate/2` is an antisymmetric, functional, injective and irreflexive and hence the domain a successor domain. The transitive closure of this domain can then be represented by natural numbers and distances easily calculated. Much like the in order heuristics, a relative distance interpretation of the above example could look like this:

```
1  eval(relative_dist(goal(inky, 100)), S, V) :-
2    fluent_holds(location(pacman, XP, YP), S),
3    fluent_holds(location(inky, XI, YI), S),
4    term_to_int(location_3_arg_2, XP, XPI),
5    term_to_int(location_3_arg_2, XI, XII),
6    domain_size(location_3_arg_2, XSize),
7    XDist is abs(XPI-XII)/XSize,
8    term_to_int(location_3_arg_3, YP, YPI),
9    term_to_int(location_3_arg_3, YI, YII),
10   domain_size(location_3_arg_3, YSize),
11   YDist is abs(YPI-YII)/YSize,
12   metric(XDist, YDist, Dist),
13   V is 1-Dist.
```

Basically, we evaluate the distances per argument, combine these using a metric and give a higher return for lower distances.

There are also predicates that do not represent a functional or injective relation. In this case we cannot construct a mapping of the domain elements of the binary static predicate to the natural numbers. To use the predicate information anyway, we construct a graph where all relations given by the predicate are considered edges of the graph. The shortest path in the graph between two arbitrary domain elements represents the distance. If there is no path, then we consider the distance infinite. Instead of using the domain size to normalize the distance, we then have to use the longest of all shortest distances between any two domain elements.

Note that we may also detect information regarding the direction: The relation expressed by `a_predicate(Y, Z)` represents, in fact, a directed edge from `Y` to `Z`, assuming that `Z` occurs in the fluent in the head of the next rule and `Y` in the body. This means that in the above evaluation lines 7 and 11 are only right if Pac-Man and Inky can move north and south. In case, they can only move in one direction, the distance is either infinite or the absolute of the determined relative distance.

Finally, the above evaluation only works if the fluents containing variables unify with exactly one instance in each state. In case there is no such fluent in the state, the distance is infinite and the result therefore $V = 0$. If there are more instances, we can calculate the pairwise relative distances and aggregate them using the average or the minimum.

The advantage of this feature are again a reduction of the size of the evaluation tree and a much greater expressiveness: Both the uninstantiated and the ground-instantiated version of e.g. the goal in Pac-Man return a constant value and thus provides no feedback in cases where Pac-Man and Inky are not on the same square. Only a relative distance evaluation returns useful fuzzy values in this case.

Though theoretically feasible in other approaches, distances between variable fluents were mentioned directly only in (Kuhlmann, Dresner, and Stone 2006) where it is used as a candidate feature. However, it remains unclear when he chooses to use the feature and how he uses the feature output in the evaluation function. In contrast, both is taken care of in our approach since we derive it from a specified expression and integrate its output via fuzzy logic.

3.2 Fluent Features

Another type of features that we identify are fluent features. This type of feature is derived from fluents and modifies an existing **true**(Fluent) expression: The standard fluent_holds/2 evaluation is substituted by a more complicated procedure, rendering the evaluation typically more expensive but also more fine-grained.

Relative Distances II - Distance To Fixed Fluents A specialization of the relative distance between two fluents is the relative distance towards a fixed point. In this pattern we do not find two fluents with shared variables but a single fluent without variables. The predicate timeout/0 is part of the goal condition of a version of MummyMaze ("mummymaze2p-comp2007"):

```
timeout :- true(step(50)).
```

The step fluent argument is increased via a static successor predicate that we identify as described in Relative Distances I. Hence the evaluation in this case is just a simplified version of the evaluation for relative distances between fluents. Another example could be a modified Pac-Man where Pac-Man has to reach a specific coordinate to win the match, e.g.

```
goal(pacman, 100) :-
  true(location(pacman, 8, 8)).
```

In fact, we can generalize this pattern to all fluents, such that for every fluent that is evaluated by fluent_holds/2 we may try to determine whether we find an order over some of its arguments. If confirmed, we use distance estimation on the fluent. The strength of the distance estimation lies in the fact that although only location(pacman, 8, 8) occurs as goal in the goal condition, we can derive useful information from *arbitrary* fluents location(pacman, X, Y) holding in the current state. The evaluation is based on the hypothesis that fluents where $X \approx 8$ and $Y \approx 8$ are more likely to lead to the desired goal fluent.

Since ground fluents occur frequently in virtually every goal condition, we must however be aware of side effects. While relative distances in Tic-Tac-Toe pose no problem as the cell coordinates are not connected via binary successor predicates, the situation is different in a Connect Four variant where the following next rule connects vertically adjacent cells:

```
next(cell(X, Y2, red)) :-
  does(red, drop(X),
  cell(X, Y2, blank),
  succ(Y1, Y2),
  cell(X, Y1, red).
```

The correct interpretation of the rule states that if a cell is occupied by red, the above cell can be occupied by red as well. However, the distance interpretation may see player red move from cell (X, Y1) to cell (X, Y2) just as Pac-Man does. The difference by which we can distinguish both interpretations lies in the fact that in Connect Four all reachable ground fluents of the form cell(X, Y, red) occur in the goal condition (as part of the definition of a line of four discs), while in a game with a distant goal there are only a few fluents that represent the goal, most do not occur in the goal condition.

A second problem arises for fluents with variables. In the example of Pawn Whopping ("pawn_whopping") the player wins if one of his pawns (symbolized by an x) reaches any cell (variable Y) of the 8^{th} rank, symbolized by the 8.

```
goal(x, 100) :- true(cell(Y, 8, x)).
```

Again, we find there are static successor predicates that impose an order over the first two arguments of the fluent, enabling us to calculate distances. However, here we have a conflict between features, as the expression could be evaluated using the solution cardinality interpretation and the distance interpretation. Of course, only distance interpretation makes sense since the goal is to have a pawn at the 8th rank and the match ends once the goal is achieved. A solution cardinality of 2 will thus never occur.

We distinguish both patterns generally by applying the solution cardinality only if all ordered arguments (that is, arguments of the fluent where an underlying order could be found) of the fluent are variables since such an expression typically counts the number of pieces on a board. In all other situations we use the distance interpretation.

Persistence Another detail for improving the evaluation of ground-instantiated fluents is their persistence. Consider the following example from Tic-Tac-Toe:

```
next(cell(X, Y, C)) :-
  true(cell(X, Y, C)), C \= b.
```

The rule states that once a cell is marked (with an x or an o), the cell remains marked for the rest of the match. We call this fluent feature "persistence" as the fact encoded by the fluent persists through the rest of the match. We distinguish between persistent true and persistent false fluents: Once a persistent true fluent holds, it holds in all successor states. In contrast, if a persistent false fluent does not hold in the current state, then it also does not hold in its successor states. An example for a persistent false fluent is the fluent cell(X, Y, b) representing unmarked (blank) cells in Tic-Tac-Toe.

Persistence can be used in several ways to improve the evaluation function: First, persistent fluents have a higher impact on a future state than non-persistent fluents. Hence, a higher evaluation is justified.

```
eval(true(Fluent), S, V) :-
  fluent_holds(Fluent, S), !,
  (persistent_true(Fluent) -> true(V);
  fz_true(V)).
eval(true(Fluent), _, V) :-
  (persistent_false(Fluent) -> false(V);
  fz_false(V)).
```

Beside persistent fluents being more stable than their non-persistent counterparts, persistence can also speed up the evaluation of a state: Once a persistent false fluent holds, any conjunction with this fluent as positive (i.e. non-negated) conjunct is also persistent false. Therefore, we can skip evaluating other conjuncts in the same conjunction. This same effect holds for negative occurrences of persistent true fluents.

Therefore we modify the evaluation function as follows:

```
eval(and([]), S, V) :- true(V).
eval(and([C|Children], S, V) :-
  eval(C, S, V1),
  (false(V1) ->
    V = 0
  ;
    eval(and(Children), S, V2),
    fz_and(V1, V2, V)
  ).
```

Naturally the idea is analogously applicable for persistent true fluents in disjunctions.

Note that distance estimations that return an infinite distance are also persistent false fluents. For proving that fluents are persistent we use the approach presented in (Thielscher and Voigt 2010).

4 Evaluation

We evaluate the effectiveness of each feature proposed by applying them in a number of games. Our hypothesis is that they increase the probability of winning against a benchmark player at least in some games. As candidate games we used only games played in the GGP championships 2005-2009. The players had a 60 seconds preparation time and 10 seconds for each move. After half of the matches, roles were switched (e.g. the party playing black plays white and vice versa) to eliminate advantages of specific roles. Both agents use the same search algorithm (α-β search).

As evaluation function we use a neural network that correctly represents propositional logic and presents similar fuzzy properties as e.g. t-norm fuzzy logic (Michulke and Thielscher 2009). We further limited the evaluation function structure to depth 8 and size 500 to cut off the most disadvantageous parts of the function. The rationale behind is that nodes in depths higher than 8 have little impact on the state evaluation, but are still expensive to evaluate. For the same reason we skip the expansion of expressions if the resulting function would surpass 500 nodes. The values of the limits were determined empirically and ensure a reasonable evaluation speed of several hundred states per second. Any remaining unexpanded expression was delegated to the reasoner.

The evaluation was performed by setting up a player against a handicapped version of itself. The handicap was realized by deactivating one of the features discussed in this paper. Both players ran on the same machine, a Core 2 Duo E 8500 at 3.16GHz with 4GB RAM, each player had 1.5GB RAM to its avail.

The left chart of Figure 1 shows the results of 40 matches in the given games. The capital letters indicate what feature was turned off at the handicapped player and refer to the expression features *solution cardinality* (SC), *order* (Ord) and *distance between fluents* (DistBtw), and the fluent features *distance towards a fixed fluent* based on natural numbers (ND) or graphs (GD), and *persistence* (Pers). The length of the bar shows to what extent the win rate was shifted in favor

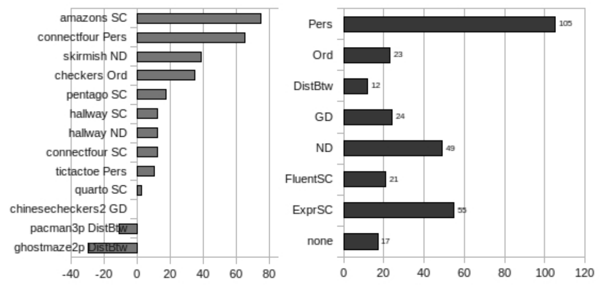

Figure 1: Left: Win Rate Increase against Handicapped Player, Right: #Features Detected in 198 Games

of the standard player when playing against the handicapped version. E.g. a value of 20% means that instead of winning 50% of all won points in the matches, the standard version now wins 60%. Consequently, the handicapped version just won 40 %.

For comparison, the chance of flipping 40 ideal coins and getting 26 times or more heads is 4.06%. So there is 4.06% chance that a win rate increase of $26/20 - 1 = 30\%$ is a mere coincidence and both agents play equally well.

We can see that in most of the games the win rate increases against the handicapped agent. The decreasing performance in the worst three games is distance related and has a simple reason: Depending on the type of search, distance information can be obtained otherwise. If e.g. our evaluation function has no distance information in Pac-Man, then all states are evaluated equal. Our architecture therefore uses the maximum of all values reachable within the search horizon to tie-break the situation. This means that once a goal state is found (even though it would require the opponent to play in our favor) the state evaluation tends towards this state. Therefore in games of limited complexity this tie-breaking mechanism in combination with the additional computational effort to calculate the distance is responsible for the underperformance. This argument is supported by the fact that in more complex games distance actually does make a difference.

The right side of Figure 1 shows in how many games features of the given type appear. ExprSC and FluentSC here distinguish between solution cardinality applied to expressions passed to the reasoner and those applied to fluents with variables[1]. We can see that there were 17 games were no feature was found. Among these are four Tic-Tac-Toe variants where persistence could not be proved within the given amount of time. All other games without features had ground goals (e.g. four Nim versions and four Blocksworld versions). Note that in these cases our fuzzy logic itself already provides good search guidance.

5 Summary

We presented a general and integrated method of how to transform a predicate logic expression to an evaluation func-

[1]The complete set of games can be found under http://euklid.inf.tu-dresden.de:8180/ggpserver

tion. We focused on detecting features in the expression, describing other conditions that must be met to use the features and proposing a number of short algorithms that show how to use the feature information. In contrast to other general game players, no expensive feature evaluation phase was needed.

We showed the benefits of applying such features on a set of specific games and measured, how general the features are on a set of 198 games submitted to the Dresden GGP server up to January 2011.

For the future, we consider it crucial to also measure the effects of the features on run-time and decide, depending on the complexity of the game, whether the feature is advantageous or not. Besides, we believe that an architecture that dynamically decides what interpretation to use is maybe suited best for the different interpretations. An idea in this scenario would be to see first whether a fluent or an expression holds. If it holds, return a value representing how often it holds and how stable (i.e. persistent) it is. If it doesn't, evaluate a fluent using distances and an expression using partial evaluation based on e.g. how many conjuncts of the expression hold. Finally, there is no reason to assume that distances cannot be calculated on the arguments of predicates, leaving space for further improvement.

References

Clune, J. 2007. Heuristic evaluation functions for general game playing. In *AAAI*. Vancouver: AAAI Press.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI*. AAAI Press.

Finnsson, H., and Björnsson, Y. 2010. Learning simulation control in general game-playing agents. In Fox, M., and Poole, D., eds., *AAAI*. AAAI Press.

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*, 1457–62. Boston, Massachusetts, USA: AAAI Press.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General game playing: Game description language specification. Technical Report March 4. The most recent version should be available at http://games.stanford.edu/.

Michulke, D., and Thielscher, M. 2009. Neural networks for state evaluation in general game playing. In *ECML PKDD '09: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 95–110. Berlin, Heidelberg: Springer-Verlag.

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *Proceedings of the National Conference on Artificial Intelligence*, 1191–1196. Vancouver: AAAI Press.

Thielscher, M., and Voigt, S. 2010. A temporal proof system for general game playing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1000–1005. Atlanta: AAAI Press.