Admissible Distance Heuristics for General Games

Daniel Michulke[1] and Stephan Schiffel[2]

[1] Dresden University of Technology, Dresden, Germany
[2] Reykjavík University, Reykjavík, Iceland
dmichulke@gmail.com, stephans@ru.is

Abstract. A general game player is a program that is able to play arbitrary games well given only their rules. One of the main problems of general game playing is the automatic construction of a good evaluation function for these games. Distance features are an important aspect of such an evaluation function, measuring, e.g., the distance of a pawn towards the promotion rank in chess or the distance between Pac-Man and the ghosts.

However, current distance features for General Game Playing are often based on too specific detection patterns to be generally applicable, and they often apply a uniform Manhattan distance regardless of the move patterns of the objects involved. In addition, the existing distance features do not provide proven bounds on the actual distances.

In this paper, we present a method to automatically construct distance heuristics directly from the rules of an arbitrary game. The presented method is not limited to specific game structures, such as Cartesian boards, but applicable to all structures in a game. Constructing the distance heuristics from the game rules ensures that the construction does not depend on the size of the state space, but only on the size of the game description which is exponentially smaller in general. Furthermore, we prove that the constructed distance heuristics are admissible, i.e., provide proven lower bounds on the actual distances.

We demonstrate the effectiveness of our approach by integrating the distance heuristics in an evaluation function of a general game player and comparing the performance with a state-of-the-art player.

Keywords: General game playing, Feature construction, Heuristic search.

1 Introduction

While in classical game playing, human experts encode their knowledge into features and parameters of evaluation functions (e.g., weights), the goal of General Game Playing is to develop programs that are able to autonomously derive a good evaluation function for a game given only the rules of the game. Because the games are unknown beforehand, the main problem lies in the detection and construction of useful features and heuristics for guiding search in the match.

One class of such features are distance features used in a variety of GGP agents (e.g., [6,9,2,4]). The way of detecting and constructing features in current game playing systems, however, suffers from a number of disadvantages:

- Distance features require a prior recognition of board-like game elements. Current approaches formulate hypotheses about which element of the game rules describes a board and then either check these hypotheses in internal simulations of the game (e.g., [6,9,4]) or try to prove them [10]. Both approaches are expensive and can only detect boards if their description follows a certain syntactic pattern.
- Distance features are limited to Cartesian board-like structures, that is, n-dimensional structures with totally ordered coordinates. Distances over general graphs are not considered.
- Distances are calculated using a predefined metric on the boards. Consequently, distance values obtained do not depend on the type of piece involved. For example, using a predefined metric the distance of a rook, king and pawn from $a2$ to $c2$ would appear equal while a human would identify the distance as 1, 2 and ∞ (unreachable), respectively.

In this paper we will present a more general approach for the construction of distance features for general games. The underlying idea is to analyze the rules of game in order to find dependencies between the fluents of the game, i.e., between the atomic properties of the game states. Based on these dependencies, we define a distance function that computes an admissible estimate for the number of steps required to make a certain fluent true. This distance function can be used as a feature in search heuristics of GGP agents. In contrast to previous approaches, our approach does not depend on syntactic patterns and involves no internal simulation or detection of any predefined game elements. Moreover, it is not limited to board-like structures but can be used for every fluent of a game.

The remainder of this paper is structured as follows: In the next section we give an introduction to the Game Description Language (GDL), which is used to describe general games. In Section 3 we introduce the theoretical basis for this work, so called fluent graphs, and show how to use them to derive distances from states to fluents. We proceed in Section 4 by showing how fluent graphs can be constructed from a game description and demonstrate their application in Section 5. We conduct experiments in Section 6 to show the benefit and generality of our approach and discuss related approaches in Section 7. Finally, we give an outlook on future work in Section 8 and summarize in Section 9.

2 Preliminaries

The language used for describing the rules of general games is the Game Description Language [7] (GDL). 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 upper-case letters stand for variables and the remaining words are constants.

```
1  role(xplayer). role(oplayer).
2
3  init(control(xplayer)).
4  init(cell(1,1,b)). init(cell(1,2,b)).init(cell(1,3,b)).
5   ...
6  init(cell(3,3,b)).
7
8  legal(P, mark(X,Y)) :- true(control(P)), true(cell(X,Y,b)).
9  legal(P, noop) :- role(P), not true(control(P)).
10
11 next(cell(X,Y,x)) :- does(xplayer, mark(X,Y)).
12 next(cell(X,Y,o)) :- does(oplayer, mark(X,Y)).
13 next(cell(X,Y,C)) :- true(cell(X,Y,C)), distinct(C, b).
14 next(cell(X,Y,b)) :- true(cell(X,Y,b)), does(P, mark(M,N)),
15   (distinct(X,M) ; distinct(Y,N)).
16
17 goal(xplayer, 100) :- line(x).
18 ...
19 terminal :- line(x) ; line(o) ; not open.
20
21 line(P) :-
22   true(cell(X,1,P)), true(cell(X,2,P)), true(cell(X,3,P)).
23 ...
24 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 8-9 define the legal moves of the game with the help of the keyword **legal**. For example, 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[1]. Lines 17 to 19 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. Fluents are the building blocks of states in GDL and will be in the center of our analysis for distance features. In the above example, there are two different types of fluents, control(X) with X $\in$ {xplayer, oplayer} and cell(X, Y, Z) with X, Y $\in$ {1, 2, 3} and Z $\in$ {b, x, o}.

In [11], we defined a formal semantics of a game described in GDL as a state transition system:

Definition 1 (Game). *Let Σ be a set of ground terms and 2^{Σ} denote the set of finite subsets of Σ. A game over this set of ground terms Σ is a state transition system $\Gamma = (R, s_0, T, l, u, g)$ over sets of states $\mathcal{S} \subseteq 2^{\Sigma}$ and actions $\mathcal{A} \subseteq \Sigma$ with*

[1] The special predicate **distinct**(X,Y) holds if the terms X and Y are syntactically different.

- $R \subseteq \Sigma$, *a finite set of roles;*
- $s_0 \in \mathcal{S}$, *the initial state of the game;*
- $T \subseteq \mathcal{S}$, *the set of terminal states;*
- $l : R \times \mathcal{A} \times \mathcal{S}$, *the legality relation;*
- $u : (R \mapsto \mathcal{A}) \times \mathcal{S} \to \mathcal{S}$, *the transition or update function; and*
- $g : R \times \mathcal{S} \mapsto \mathbb{N}$, *the reward or goal function.*

This formal semantics is based on a set of ground terms Σ. This set is the set of all ground terms over the signature of the game description. Hence, fluents, actions and roles of the game are ground terms in Σ. States are finite sets of fluents, i.e., finite subsets of Σ. The connection between a game description D and the game Γ it describes is established using the standard model of the logic program D. For example, the update function u is defined as

$$u(A, s) = \{f \in \Sigma : D \cup s^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{next}(f)\}$$

where $s^{\texttt{true}}$ and $A^{\texttt{does}}$ are suitable encodings of the state s and the joint action A of all players as a logic program. Thus, the successor state $u(A, s)$ is the set of all ground terms (fluents) f such that $\texttt{next}(f)$ is entailed by the game description D together with the state s and the joint move A. For a complete definition for all components of the game Γ we refer to [11].

3 Fluent Graphs

Our goal is to obtain knowledge on how fluents evolve over time. We start by building a *fluent graph* that contains all the fluents of a game as nodes. Then we add directed edges (f_i, f) if at least one of the predecessor fluents f_i must hold in the current state for the fluent f to hold in the successor state. Figure 1(a) shows a partial fluent graph for Tic-Tac-Toe that relates the fluents `cell(3,1,Z)` for `Z` $\in$ `{b, x, o}`.

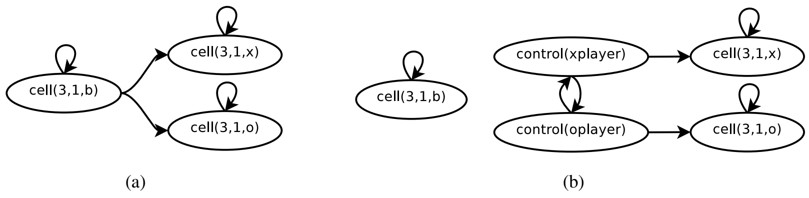

Fig. 1. Shown are two possible partial fluent graphs for Tic-Tac-Toe, where (a) captures the dependencies between different markers on a cell while (b) fails to capture these dependencies

For cell `(3,1)` to be blank it had to be blank before. For a cell to contain an `x` (or an `o`) in the successor state there are two possible preconditions. Either, it contained an `x` (or `o`) before or it was blank.

Using this graph, we can conclude that, e.g., a transition from `cell(3,1,b)` to `cell(3,1,x)` is possible within one step while a transition from `cell(3,1,o)` to `cell(3,1,x)` is impossible.

To build on this information, we formally define a fluent graph as follows:

Definition 2 (Fluent Graph). *Let Γ be a game over ground terms Σ. A graph $G = (V, E)$ is called a fluent graph for Γ iff*

- *$V = \Sigma \cup \{\emptyset\}$ and*
- *for all fluents $f \in \Sigma$, two valid states s and s'*

$$\begin{aligned} &(s' \textit{ is a successor of } s) \wedge f' \in s' \\ &\quad \Rightarrow (\exists f)(f, f') \in E \wedge (f \in s \cup \{\emptyset\}) \end{aligned} \tag{1}$$

In this definition we add an additional node $\emptyset$ to the graph and allow $\emptyset$ to occur as the source of edges. The reason is that there can be fluents in the game that do not have any preconditions, for example the fluent `g` with the following next rule: `next(g) :- distinct(a,b)`. On the other hand, there might be fluents that cannot occur in any state, because the body of the corresponding next rule is unsatisfiable, for example: `next(h) :- distinct(a,a)`. We distinguish between fluents that have no precondition (such as g) and fluents that are unreachable (such as h) by connecting the former to the node $\emptyset$ while unreachable fluents have no edge in the fluent graph.

Note that the definition covers only some of the necessary preconditions for fluents, therefore fluent graphs are not unique as Figure 1(b) shows. We will address this problem later.

We can now define a distance function $\Delta(s, f')$ between the current state s and a state in which fluent f' holds as follows:

Definition 3 (Distance Function). *Let $\Delta_G(f, f')$ be the length of the shortest path from node f to node f' in the fluent graph G or ∞ if there is no such path. Then we define*

$$\Delta(f, f') \overset{\text{def}}{=} \begin{cases} 0 & f = f' \\ \Delta_G(f, f') & \textit{else} \end{cases}$$

$$\Delta(s, f') \overset{\text{def}}{=} \min_{f \in s \cup \{\emptyset\}} \Delta(f, f')$$

That means, the distance $\Delta(s, f')$ is 0 if and only if f' holds in s, otherwise it is compute as the shortest path in the fluent graph from any fluent in s to f'.

Intuitively, each edge (f, f') in the fluent graph corresponds to a state transition of the game from a state in which f holds to a state in which f' holds. Thus, the length of a path from f to f' in the fluent graph corresponds to the number of steps in the game between a state containing f to a state containing f'. Of course, the fluent graph is an abstraction of the actual game: many preconditions for the state transitions are ignored. As a consequence, the distance $\Delta(s, f')$ that we compute in this way is a lower bound on the actual number of steps it takes to go from s to a state in which f' holds. Therefore the distance $\Delta(s, f')$ is an admissible heuristic for reaching f' from a state s.

Theorem 1 (Admissible Distance). *Let*
- *$\Gamma = (R, s_0, T, l, u, g)$ be a game with ground terms Σ and states $\mathcal{S}$,*
- *$s_1 \in \mathcal{S}$ be a state of Γ,*

– *$f \in \Sigma$ be a fluent of Γ, and*
– *$G = (V, E)$ be a fluent graph for Γ.*

Furthermore, let $s_1 \mapsto s_2 \mapsto \ldots \mapsto s_{m+1}$ denote a legal sequence of states of Γ, that is, for all i with $0 < i \leq m$ there is a joint action A_i, such that:

$$s_{i+1} = u(A_i, s_i) \wedge (\forall r \in R) l(r, A_i(r), s)$$

If $\Delta(s_1, f) = n$, then there is no legal sequence of states $s_1 \mapsto \ldots \mapsto s_{m+1}$ with $f \in s_{m+1}$ and $m < n$.

Proof. We prove the theorem by contradiction. Assume that $\Delta(s_1, f) = n$ and there is a legal sequence of states $s_1 \mapsto \ldots \mapsto s_{m+1}$ with $f \in s_{m+1}$ and $m < n$. By Definition 2, for every two consecutive states s_i, s_{i+1} of the sequence $s_1 \mapsto \ldots \mapsto s_{m+1}$ and for every $f_{i+1} \in s_{i+1}$ there is an edge $(f_i, f_{i+1}) \in E$ such that $f_i \in s_i$ or $f_i = \emptyset$. Therefore, there is a path $f_j, \ldots, f_m, f_{m+1}$ in G with $1 \leq j \leq m$ and the following properties:

– $f_i \in s_i$ for all $i = j, ..., m + 1$,
– $f_{m+1} = f$, and
– either $f_j \in s_1$ (e.g., if $j = 1$) or $f_j = \emptyset$.

Thus, the path $f_j, \ldots, f_m, f_{m+1}$ has a length of at most m.

Consequently, $\Delta(s_1, f) \leq m$ because $f_j \in s_1 \cup \{\emptyset\}$ and $f_{m+1} = f$. However, $\Delta(s_1, f) \leq m$ together with $m < n$ contradicts $\Delta(s_1, f) = n$. □

4 Constructing Fluent Graphs from Rules

We propose an algorithm to construct a fluent graph based on the rules of the game. The transitions of a state s to its successor state s' are encoded fluent-wise via the **`next`** rules. Consequently, for each $f' \in s'$ there must be at least one rule with the head **`next`**`(f')`. All fluents occurring in the body of these rules are possible sources for an edge to f' in the fluent graph.

For each ground fluent f' of the game:

1. Construct a ground disjunctive normal form ϕ of `next`(f'), i.e., a formula ϕ such that `next`$(f') \supset \phi$.
2. For every disjunct ψ in ϕ:
   – Pick one literal `true`(f) from ψ or set $f = \emptyset$ if there is none.
   – Add the edge (f, f') to the fluent graph.

Note, that we only select one literal from each disjunct in ϕ. Since, the distance function $\Delta(s, f')$ obtained from the fluent graph is admissible, the goal is to construct a fluent graph that increases the lengths of the shortest paths between the fluents as much as possible. Therefore, the fluent graph should contain as few edges as possible. In general the complete fluent graph (i.e., the graph where every fluent is connected to every other fluent) is the least informative because the maximal distance obtained from this graph is 1.

The algorithm outline still leaves some open issues:

1. How do we construct a ground formula ϕ that is the disjunctive normal form of $\texttt{next}(f')$?
2. Which literal $\texttt{true}(f)$ do we select if there is more than one? Or, in other words, which precondition f' of f do we select?

We will discuss both issues in the following sections.

4.1 Constructing a DNF of $\texttt{next}(\mathbf{f'})$

A formula ϕ in DNF is a set of formulas $\{\psi_1, \dots, \psi_n\}$ connected by disjunctions such that each formula ψ_i is a set of literals connected by conjunctions. We propose the algorithm in Figure 1 to construct ϕ such that $\texttt{next}(f') \supset \phi$.

Algorithm 1. Constructing a formula ϕ in DNF with $\texttt{next}(f') \supset \phi$

```
Input: game description $D$, ground fluent $f'$
Output: $\phi$, such that $\texttt{next}(f') \supset \phi$
 1: $\phi := \texttt{next}(f')$
 2: $finished := false$
 3: while $\neg finished$ do
 4:    Replace every positive occurrence of $\texttt{does}(r, a)$ in $\phi$ with $\texttt{legal}(r, a)$.
 5:    Select a positive literal $l$ from $\phi$ such that $l \neq \texttt{true}(t), l \neq \texttt{distinct}(t_1, t_2)$ and $l$ is
       not a recursively defined predicate.
 6:    if there is no such literal then
 7:       $finished := true$
 8:    else
 9:       $\hat{l} := \bigvee_{h:-b \in D, l\sigma = h\sigma} b\sigma$
10:       $\phi := \phi\{l/\hat{l}\}$
11:    end if
12: end while
13: Transform $\phi$ into disjunctive normal form, i.e., $\phi = \psi_1 \vee \ldots \vee \psi_n$ and each formula $\psi_i$ is a
    conjunction of literals.
14: for all $\psi_i$ in $\phi$ do
15:    Replace $\psi_i$ in $\phi$ by a disjunction of all ground instances of $\psi_i$.
16: end for
```

The algorithm starts with $\phi = \texttt{next}(f')$. Then, it selects a positive literal l in ϕ and unrolls this literal, that is, it replaces l with the bodies of all rules $h\,{:}\text{-}\,b \in D$ whose head h is unifiable with l with a most general unifier σ (lines 9, 10). The replacement is repeated until all predicates that are left are either of the form $\texttt{true}(t)$, $\texttt{distinct}(t_1, t_2)$ or recursively defined. Recursively defined predicates are not unrolled to ensure termination of the algorithm. Finally, we transform ϕ into disjunctive normal form and replace each disjunct ψ_i of ϕ by a disjunction of all of its ground instances in order to get a ground formula ϕ.

Note that in line 4, we replace every occurrence of **`does`** with **`legal`** to also include the preconditions of the actions that are executed in ϕ. As a consequence the resulting formula ϕ is not equivalent to $\texttt{next}(f')$. However, $\texttt{next}(f') \supset \phi$, under the assumption that only legal moves can be executed, i.e., $\texttt{does}(r, a) \supset \texttt{legal}(r, a)$. This is sufficient for constructing a fluent graph from ϕ.

Note, that we do not select negative literals for unrolling because the construction of our fluent graph only requires positive preconditions for fluents. Still, the algorithm could be easily adapted to also unroll negative literals at the cost of an increased size of the created ϕ.

4.2 Selecting Preconditions for the Fluent Graph

If there are several literals of the form $\texttt{true}(f)$ in a disjunct ψ of the formula ϕ constructed above, we have to select one of them as source of the edge in the fluent graph. As already mentioned, the distance $\Delta(s, f)$ computed with the help of the fluent graph is a lower bound on the actual number of steps needed. To obtain a good lower bound, that is, one as large as possible, the paths between nodes in the fluent graph should be as long as possible. Selecting the best fluent graph, i.e., the one which maximizes the distances, is impossible since this depends on the states we encounter when playing the game, and we do not know these states beforehand. In order to generate a fluent graph that provides good distance estimates, we use several heuristics when we select literals from disjuncts in the DNF of $\texttt{next}(f')$:

First, we only add new edges if necessary. That means, whenever there is a literal $\texttt{true}(f)$ in a disjunct ψ such that the edge (f, f') already exists in the fluent graph, we select this literal $\texttt{true}(f)$. The rationale of this heuristic is that paths in the fluent graph are longer on average if there are fewer connections between the nodes.

Second, we prefer a literal $\texttt{true}(f)$ over $\texttt{true}(g)$ if f is more similar to f' than g is to f', that is $sim(f, f') > sim(g, f')$.

We define the similarity $sim(t, t')$ recursively over ground terms t, t':

$$sim(t,t') \overset{\text{def}}{=} \begin{cases} 1 & t, t' \text{ have arity 0 and } t = t' \\ \sum_i sim(t_i, t_i') & t = f(t_1, \ldots, t_n) \text{ and} \\ & t' = f(t_1', \ldots, t_n') \\ 0 & else \end{cases}$$

In human made game descriptions, similar fluents typically have strong connections. For example, in Tic-Tac-Toe `cell(3,1,x)` is more related to `cell(3,1,b)` than to `cell(b,3,x)`. By using similar fluents when adding new edges to the fluent graph, we have a better chance of finding the same fluent again in a different disjunct of ϕ. Thus we maximize the chance of reusing edges.

5 Applying Distance Features

For using the distance function in our evaluation function, we define the normalized distance $\delta(s, f)$.

$$\delta(s, f) \overset{\text{def}}{=} \frac{\Delta(s, f)}{\Delta_{max}(f)}$$

The value $\Delta_{max}(f)$ represents the longest *finite* distance $\Delta_G(g, f)$ from any fluent g to f in G.

Thus the normalized distance $\delta(s, f)$ will be infinite if and only if $\Delta(s, f) = \infty$, i.e., there is no path from any fluent in s to f in the fluent graph. In all other cases it holds that $0 \leq \delta(s, f) \leq 1$.

Note, that the construction of the fluent graph and computing the shortest paths between all fluents, i.e., the distance function Δ_G, need only be done once for a game. Thus, while construction of the fluent graph is more expensive for complex games, the cost of computing the distance feature $\delta(s, f)$ (or $\Delta(s, f)$) only depends (linearly) on the size of the state s.

5.1 Using Distance Features in an Evaluation Function

To demonstrate the application of the distance measure presented, we use a simplified version of the evaluation function of Fluxplayer [9] implemented in Prolog. It takes the ground DNF of the goal rules as first argument, the current state as second argument and returns the fuzzy evaluation of the DNF on that state as a result.

```
1 eval((D1; ...; Dn), S, R) :- !,
2   eval(D1, S, R1), ..., eval(Dn, S, Rn),
3   R is sum(R1, ..., Rn) - product(R1, ..., Rn).
4 eval((C1, ..., Cn), S, R) :- !,
5   eval(C1, S, R1), ..., eval(Cn, S, Rn),
6   R is product(R1, ..., Rn).
7 eval(not(P), S, R) :- !, eval(P, S, Rp), R is 1 - Rp.
8 eval(true(F), S, 0.9) :- occurs(F, S),!.
9 eval(true(F), S, 0.1).
```

Disjunctions are transformed to probabilistic sums, conjunctions to products, and **true** statements are evaluated to values in the interval $[0, 1]$, basically resembling a recursive fuzzy logic evaluation using the product t-norm and the corresponding probabilistic sum t-conorm. The state value increases with each conjunct and disjunct fulfilled.

We compare the evaluation to a second function that employs our relative distance measure, encoded as predicate `delta`. We obtain the distance-based evaluation function by substituting line 9 of the previous program by the following:

```
1 eval(true(F), S, R) :- delta(S, F, Dist), Dist =< 1, !,
2   R is 0.8*(1-Dist) + 0.1.
3 eval(true(F), S, 0).
```

Here, we evaluate a fluent that does not occur in the current state to a value in $[0.1, 0.9]$ and, in case the relative distance is infinite, to 0 since this means that the fluent cannot hold anymore.

5.2 Tic-Tac-Toe

Although on first sight Tic-Tac-Toe contains no relevant distance information, we can still take advantage of our distance function. Consider the two states as shown in Figure 2. In state s_1 the first row consists of two cells marked with an `x` and a blank cell.

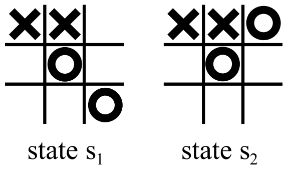

Fig. 2. Two states of Tic-Tac-Toe. The first row is still open in state s_1 but blocked in state s_2.

In state s_2 the first row contains two `x`s and one cell marked with an `o`. State s_1 has a higher state value than s_2 for `xplayer` since in s_1 `xplayer` has a threat of completing a line in contrast to s_2. The corresponding goal condition for `xplayer` completing the first row is:

```
line(x) :- true(cell(1,1,x)),
  true(cell(2,1,x)), true(cell(3,1,x)).
```

When evaluating the body of this condition using the above fuzzy evaluation, we see that it cannot distinguish between s_1 and s_2 because both have two markers in place and one missing for completing the line for `xplayer`, resulting in an evaluation $\texttt{R} = 1 * 1 * 0.1 = 0.1$

However, the distance-based function evaluates **`true`**`(cell(3,1,b))` of s_1 to 0.1 and **`true`**`(cell(3,1,o))` of s_2 to 0. Therefore, it can distinguish between both states, returning $\texttt{R} = 0.1$ for $\texttt{S} = s_1$ and $\texttt{R} = 0$ for $\texttt{S} = s_2$.

5.3 Breakthrough

The second game is Breakthrough, again a two-player game played on a chessboard. Like in chess, the first two ranks contain only white pieces and the last two only black pieces. The pieces of the game are only pawns that move and capture in the same way as pawns in chess, but without the initial double advance. Whoever reaches the opposite side of the board first wins. [2] Figure 3 shows the initial position for Breakthrough. The arrows indicate the possible moves, a pawn can make.

The goal condition for the player `black` states that black wins if there is a cell with the coordinates `X,1` and the content `black`, such that `X` is an index (a number from 1 to 8 according to the rules of `index`):

```
goal(black, 100) :- index(X),true(cellholds(X, 1, black)).
```

Grounding this rule yields

```
goal(black, 100) :- true(cellholds(1, 1, black) ;
  ...; true(cellholds(8, 1, black)).
```

We omitted the `index` predicate since it is true for all 8 ground instances.

[2] The complete rules for Breakthrough as well as Tic-Tac-Toe can be found under `http://ggpserver.general-game-playing.de/`

The standard evaluation function cannot distinguish any of the states in which the goal is not reached because **true**(cellholds(X, 1, black)) is false in all of these states for any instance of X.

The distance-based evaluation function is able to construct a fluent graph as depicted in Figure 4 for distance calculation.

Therefore evaluations of atoms of the form **true**(cellholds(X, Y, black)) have now 9 possible values (for distances 0 to 7 and ∞) instead of 2 (true and false). Hence, states where black pawns are nearer to one of the cells (1,8), …, (8,8) are preferred.

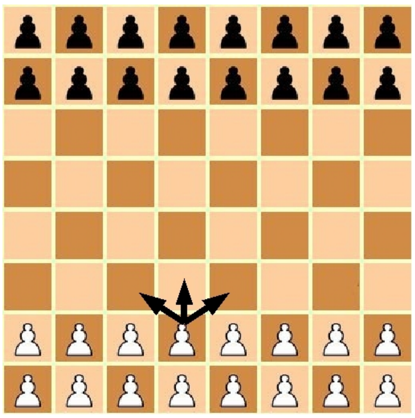

Fig. 3. Initial position in Breakthrough and the move options of a pawn

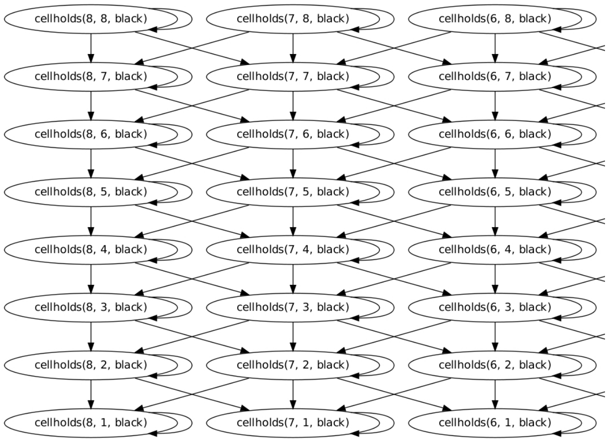

Fig. 4. A partial fluent graph for Breakthrough, role black

Moreover, the fluent graph, and thus the distance function, reflect what could be called “strategic positioning”: states with pawns on the side of the board are worth less than those with pawns in the center. This is due to the fact that a pawn in the center may reach more of the 8 possible destinations than a pawn on the side.

6 Evaluation

We first measure the generality and time requirements of our approach for 201 games available at http://ggpserver.general-game-playing.de/.

Figure 5 shows the number of games grouped by the minimum time limit (in seconds) required for successful fluent graph construction.

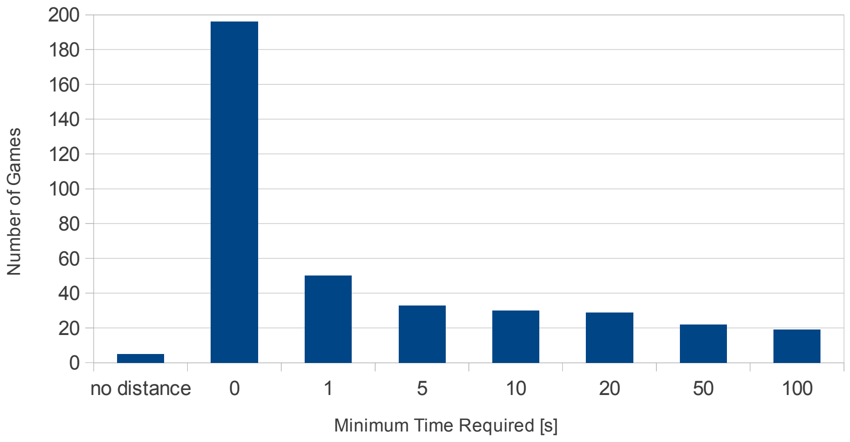

Fig. 5. Number of Games grouped by the minimum time limit required for construction of the fluent graph

We can see that the approach is able to find distance features in all but 5 games. Construction is typically fast and takes a few seconds for the majority of games. There are, however, 33 games that require at least 5 seconds for fluent graph construction and 22 of these even more than 50 seconds. Our interpretation of the results is that the approach is general, although time constraints may pose a problem in some cases and should be addressed, e.g., using time-outs.

For evaluation of the playing strength we implemented our distance function and equipped the agent system Fluxplayer [9] with it. We then set up this version of Fluxplayer (“flux_distance”) against its version without the new distance function (“flux_basic”). We used the version of Fluxplayer that came in 4th in the 2010 championship. Since flux_basic is already endowed with a distance heuristic, the evaluation is comparable to a competition setting of two competing heuristics using distance features.

We chose 19 games for comparison in which we conducted 100 matches on average. Figure 6 shows the results.

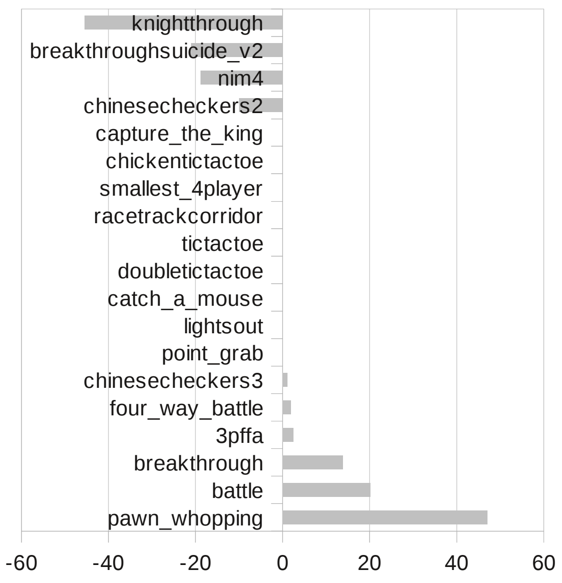

Fig. 6. Advantage in Win Rate of flux_distance

The values indicate the difference in win rate, e.g., a value of $+10$ indicates that flux_distance won 55% of the games against flux_basic winning 45%. Obviously the proposed heuristics produces results comparable to the flux_basic heuristics, with both having advantages in some games. This has several reasons: Most importantly, our proposed heuristic, in the way it is implemented now, is more expensive than the distance estimation used in flux_basic. Therefore the evaluation of a state takes longer and the search tree can not be explored as deeply as with cheaper heuristics. This accounts for three of the four underperforming games. For example in nim4, the flux_basic distance estimation provides essentially the same results as our new approach, just much cheaper. In chinesecheckers2 and knightthrough, the new distance function slows down the search more than its better accuracy can compensate.

On the other hand, flux_distance performs better in complicated games. There the higher accuracy of the heuristics typically outweighs the disadvantage of the heuristics being slower.

Interestingly the higher accuracy of the new distance heuristics is the reason for flux_distance losing in breakthrough_suicide. The game is exactly the same as breakthrough, however, the player to reach the other side of the board first does not win but loses.

The heuristics of both flux_basic and flux_distance are not good for this game since both are based the minimum number of moves necessary to reach the goal while the optimal heuristic would depend on the maximum number of moves available to avoid losing. However, since flux_distance is more accurate, flux_distance selects even worse moves that flux_basic. Specifically, flux distance tries to maximize (a much more accurate) minimal distance to the other side of the board, thereby allowing the opponent to capture its advanced pawns. This behavior, however, results in a smaller maximal number of moves until the game ends and forces to advance the few remaining pawns quickly. Thus, the problem in this game is not the distance estimate but the fact that the heuristic is not suitable for the game.

Finally, in some of the games no changes were found since both distance estimates performed equally well. However, rather specific heuristics and analysis methods of flux_basic could be replaced by our new general approach. For example, the original Fluxplayer contains a special method to detect when a fluent is unreachable, while this information is automatically included in our distance estimate.

7 Related Work

Distance features are part of classical agent programming for games like chess and checkers in order to measure, e.g., the distance of a pawn to the promotion rank. A more general detection mechanism was first employed in Metagamer [8] where the features "promote-distance" and "arrival-distance" represented a value indirectly proportional to the distance of a piece to its arrival or promotion square. However, due to the restriction on symmetric chess-like games, the features are still too specific for an application in GGP.

Currently, a number of GGP agent systems apply distance features in different forms. UTexas [6] identifies order relations syntactically and tries to find 2d-boards with coordinates ordered by these relations. Properties of the content of these cells, such as minimal/maximal x- and y-coordinates or pair-wise Manhattan distances are then assumed as candidate features and may be used in the evaluation function. Fluxplayer [9] generalizes the detection mechanism using semantic properties of order relations and extends board recognition to arbitrarily defined n-dimensional boards.

Another approach is pursued by Cluneplayer [2] who tries to impose a symbol distance interpretation on expressions found in the game description. Symbol distances, however, are again calculated using Manhattan distances on ordered arguments of board-like fluents, eventually resulting in a similar distance estimate as UTexas and Fluxplayer.

Although not explained in detail, Ogre [4] also employs two features that measure the distance from the initial position and the distance to a target position. Again, Ogre relies on syntactic detection of order relations and seems to employ a board centered metrics, ignoring the piece type.

All of these approaches rely on the identification of certain fixed structures in the game (such as game boards) but can not be used for fluents that do not belong to such a structure. Furthermore, they make assumptions about the distances on these structures (usually Manhattan distance) that are not necessarily connected to the game dynamics, e.g., how different pieces move on a board.

In domain independent planning, distance heuristics are used successfully, e.g., in HSP [1] and FF [3]. The heuristics $h(s)$ used in these systems is an approximation of the plan length of a solution in a relaxed problem, where negative effects of actions are ignored. This heuristics is known as delete list relaxation. While on first glance this may seems very similar to our approach, several differences exist:

- The underlying languages, GDL for general game playing and PDDL for planning, are different. A translation of GDL to PDDL is expensive in many games [5]. Thus, directly applying planning systems is not often not feasible.
- The delete list relaxation considers all (positive) preconditions of a fluent, while we only use one precondition. This enables us to precompute the distance between the fluents of a game.
- While goal conditions of most planning problems are simple conjunctions, goals in the general games can be very complex (e.g., checkmate in chess). Additionally, the plan length is usually not a good heuristics, given that only the own actions and not those of the opponents can be controlled. Thus, distance estimates in GGP are usually not used as the only heuristics but only as a feature in a more complex evaluation function. As a consequence, computing distance features must be relatively cheap.
- Computing the plan length of the relaxed planning problem is NP-hard, and even the approximations used in HSP or FF that are not NP-hard require to search the state space of the relaxed problem. On the other hand, computing distance estimates with our solution is relatively cheap. The distances $\Delta_G(f, g)$ between all fluents f and g in the fluent graph can be precomputed once for a game. Then, computing the distance $\Delta(s, f')$ (see Definition 3) is linear in the size of the state s, i.e., linear in the number of fluents in the state.

8 Future Work

One problem of the approach is its computational cost for constructing the fluent graph, that may prevent an application of our distance features for narrow time constraints.

One way to reduce the time needed for construction is a reduction of the size of ϕ via a more selective expansion of predicates (line 5) in Algorithm 1. Developing heuristics for this step is one of the goals for future research.

In addition, we are working on a way to construct fluent graphs from non-ground representations of the preconditions of a fluent to skip the grounding step. For example, the partial fluent graph in Figure 1(a) is identical to the fluent graphs for the other 8 cells of the Tic-Tac-Toe board. The fluent graphs for all 9 cells are obtained from the same rules for `next(cell(X,Y,_)`, just with different instances of the variables `X` and `Y`. By not instantiating `X` and `Y`, the generated DNF is exponentially smaller while still containing the same information.

The quality of the distance estimates depends mainly on the selection of preconditions. At the moment, the heuristics we use for this selection are intuitive but have no thorough theoretic or empiric foundation. In future, we want to investigate how these heuristics can be improved.

Furthermore, we intend to enhance the approach to use fluent graphs for generalizations of other types of features, such as, piece mobility and strategic positions.

9 Summary

We have presented a general method of deriving distance estimates in General Game Playing. To obtain such a distance estimate, we introduced fluent graphs, proposed an algorithm to construct them from the game rules and demonstrated the transformation from fluent graph distance to a distance feature.

Unlike previous distance estimations, our approach does not rely on syntactic patterns or internal simulations. Moreover, it preserves piece-dependent move patterns and produces an admissible distance heuristic.

We showed on an example how these distance features can be used in a state evaluation function. We gave two examples on how distance estimates can improve the state evaluation and evaluated our distance against Fluxplayer in its most recent version.

Certain shortcomings should be addressed to improve the efficiency of fluent graph construction and the quality of the obtained distance function. Despite these shortcomings, we found that a state evaluation function using the new distance estimates can compete with a state-of-the-art system.

References

1. Bonet, B., Geffner, H.: Planning as heuristic search. Artificial Intelligence 129(1-2), 5–33 (2001)
2. Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 1134–1139. AAAI Press (2007)
3. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. JAIR 14, 253–302 (2001)
4. Kaiser, D.M.: Automatic feature extraction for autonomous general game playing agents. In: Proceedings of the Sixth Intl. Joint Conf. on Autonomous Agents and Multiagent Systems (2007)
5. Kissmann, P., Edelkamp, S.: Instantiating General Games Using Prolog or Dependency Graphs. In: Dillmann, R., Beyerer, J., Hanebeck, U.D., Schultz, T. (eds.) KI 2010. LNCS, vol. 6359, pp. 255–262. Springer, Heidelberg (2010)
6. Kuhlmann, G., Dresner, K., Stone, P.: Automatic Heuristic Construction in a Complete General Game Player. In: Proceedings of the Twenty-First National Conference on Artificial Intelligence, pp. 1457–1462. AAAI Press, Boston (2006)
7. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: Game description language specification. Tech. Rep., Stanford University (March 4, 2008), the most recent version should be available at `http://games.stanford.edu/`
8. Pell, B.: Strategy generation and evaluation for meta-game playing. Ph.D. thesis, University of Cambridge (1993)
9. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the National Conference on Artificial Intelligence, pp. 1191–1196. AAAI Press, Vancouver (2007)
10. Schiffel, S., Thielscher, M.: Automated theorem proving for general game playing. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) (2009)
11. Schiffel, S., Thielscher, M.: A Multiagent Semantics for the Game Description Language. In: Filipe, J., Fred, A., Sharp, B. (eds.) ICAART 2009. CCIS, vol. 67, pp. 44–55. Springer, Heidelberg (2010)