Evaluation Functions in General Game Playing

Dissertation

zur Erlangung des akademischen Grades

Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt an der

Technischen Universität Dresden

Fakultät Informatik

eingereicht von

Daniel Michulke

geboren am 13. April 1982 in Stollberg

dmichulke@gmail.com

Gutachter:	Prof. Michael Thielscher
	Prof. Stefan Edelkamp
Datum der Einreichung:	27. April 2012
Datum der Verteidigung:	22. Juni 2012

Abstract

While in traditional computer game playing agents were designed solely for the purpose of playing one single game, General Game Playing is concerned with agents capable of playing classes of games. Given the game's rules and a few minutes time, the agent is supposed to play any game of the class and eventually win it.

Since the game is unknown beforehand, previously optimized data structures or human-provided features are not applicable. Instead, the agent must derive a strategy on its own. One approach to obtain such a strategy is to analyze the game rules and create a state evaluation function that can be subsequently used to direct the agent to promising states in the match.

In this thesis we will discuss existing methods and present a general approach on how to construct such an evaluation function. Each topic is discussed in a modular fashion and evaluated along the lines of quality and efficiency, resulting in a strong agent.

Acknowledgements

It is my pleasure to thank all the people who helped make this thesis possible.

First and foremost, I would like to thank my advisor Michael Thielscher for his guidance through all the years. He introduced me to the topic that today affects my life in areas I did not expect. His advice and feedback were at all times invaluable and his words always marked by precision and patience.

I am also indebted to Stephan Schiffel. Besides our collaboration on a few articles, his support in reviewing papers and this thesis cannot be matched. Our numerous discussions were thought provoking and often a fruitful source for new ideas, as well as a sink for those that would have led to a dead end.

I also want to thank all my other colleagues at the Dresden University for our sometimes helpful but always joyous discussions.

Finally, my gratitude goes to my friends and family who were incredibly supportive during all these years. Especially, I want to thank my partner Cíntia for her love and trust and her family who helped me to get through the last months of this thesis.

This thesis was supported by the German National Academic Foundation and I herewith would like to express my gratitude for financing and supporting this work.

Contents

1. Introduction **1**
- 1.1. Evaluation Functions . . . 1
- 1.2. Contributions . . . 2
- 1.3. Outline . . . 4

2. Game Playing **5**
- 2.1. General Game Playing . . . 6
- 2.2. Basic Notions of Games . . . 15
- 2.3. Move Selection and State Evaluation . . . 16
- 2.4. Search Algorithms . . . 21
- 2.5. Criteria of Evaluation Functions . . . 25
- 2.6. A Word on Experimental Evaluation . . . 27

3. Evaluation Functions I - Aggregation **30**
- 3.1. Choice of Evaluation Function . . . 30
- 3.2. An Aggregation Function Based on Neural Networks . . . 34
- 3.3. High-Resolution State Evaluation using Neural Networks . . . 46
- 3.4. Summary . . . 59

4. Evaluation Functions II - Features **60**
- 4.1. Categorization of Features . . . 60
- 4.2. A New View on Features . . . 69
- 4.3. Detection of Rule-Derived Features . . . 76
- 4.4. Integration of Rule-Derived Features . . . 86
- 4.5. Acquisition of Admissible Distance Features . . . 100
- 4.6. Summary . . . 111

5. General Evaluation **115**
- 5.1. Final Version of Nexplayer . . . 115
- 5.2. Past Competitions . . . 117
- 5.3. Experiments . . . 119
- 5.4. Summary . . . 121

6. Related Work **126**
- 6.1. Probabilistic Agents . . . 126
- 6.2. Deterministic Agents . . . 128
- 6.3. Summary of GGP Agents . . . 133

6.4. Non-GGP Approaches 134

7. Discussion **136**
7.1. Summary 136
7.2. Future Work 137
7.3. Publications 139

A. Appendix **140**
A.1. Evaluation Setup 140
A.2. Other Improvements 141

B. Bibliography **145**

1. Introduction

Much like the ultimate goal of the drivers of industrialization was to relieve humanity of some of the hardships imposed by practical labor, research in the area of Artificial Intelligence aims to facilitate and support mental processes performed by humans.

Following a behaviorist perspective, intelligence is a trait assigned to an agent by an observer based on the evidence of the agent's (externally observable) behavior [Ski53]. In the case of non-living agents, such as agent programs, behavior is limited to actions and generally preceded by decisions. The goal of creating Artificial Intelligence can consequently be reduced to making intelligent decisions.

An integral part of an intelligent decision is to consider the consequences, that is, derive the consequences of each possible action and compare them against each other. However, the consequences are often incomparable and thus need to be mapped to a common domain. Such mappings are typically represented by evaluation functions. Thus, by evaluating the consequences of actions and comparing the results, we can arrive at informed decisions.

A well-defined and observable yet complex domain for evaluation functions are games. Here, agents have the goal of "winning the game" by selecting in each state which move to make.

1.1. Evaluation Functions

In order to decide for or against a move, agents need to evaluate the consequences of their moves. For this purpose they employ evaluation functions that return information as to how positive a move should be regarded.

Evaluation functions thus determine to a large part the behavior of an agent and will therefore be in the focus of this work. There are, however, other reasons to study evaluation functions in games.

Most importantly, evaluation functions are much more general than their use in game playing agents suggests: Given that they estimate the value of abstract entities based on currently available evidence, they can be seen as predictor. As we will argue throughout this work, evaluation functions for game playing agents are evaluations of predictions for terminal states based on the current state. As such, they represent state value predictors based on current state, comparable to stock price predictors given the current stock market data or a meteorological forecasts based on today's weather. The chaotic dynamics in game playing stem from other player's moves and the problem of controlling a variable (the agent's long-term reward) based on weakly related short-term evidence (the current state).

Still, game playing is easier than the two aforementioned prediction problems for three reasons. First, there is an explicit domain theory available that can be used, avoiding thereby an imprecise description of the problem. Second, the domain theory is known to be complete, theoretically eliminating the necessity for empiric evidence. And third, states in the games we investigate are Markovian, meaning that states prior to the current state can be disregarded for predicting future states.

We argue that perspectives, ideas and approaches for addressing state value prediction will also provide insights for the general class of prediction problems.

1.2. Contributions

As domain of application we will use General Game Playing (GGP) which is an abstraction of traditional game playing where the game rules are only known at run time. Thus, a GGP agent cannot be adapted to the game by its programmer but must derive a strategy for each game on its own.

There are two types of evaluation functions employed by GGP agents, probabilistic and deterministic evaluation functions. We will argue in favor of deterministic evaluation functions and analyze its construction along its two components:

Features are the basic elements of evaluation function and evaluate specific aspects of a state.

The Aggregation Function then works on top of these features and aggregates the feature values to produce a single value.

The composition of these functions is the evaluation function.

The goal of this work is to provide an extensive discussion on how to construct an evaluation function. We aim to achieve this goal by discussing each component of the evaluation function with the following plan in mind:

State of the Art We analyze the establishments of previous work.

Categorization We categorize existing work to allow for a brief but comprehensive discussion.

Theoretical Evaluation We evaluate the categories along specific guidelines to see what approach is best for constructing the component.

Construction We construct the component and evaluate it.

Improvement We discuss important improvements to our construction approach and evaluate it again.

Our intention is to add value to the scientific discussion by motivating each step based on the findings in the discussion of its predecessor.

A consequence of this structured approach is that we will be able to discover interesting relationships already while categorizing and evaluating a subject theoretically. These

relationships are quite different from the conclusions drawn in GGP research dominated by practical evaluations, such as:

- a set of theoretical criteria that an evaluation function should fulfill in order promote the playing strength of its agent,
- theoretical conclusions on what types and components of evaluation functions are best-suited, motivated by the application of the above criteria,
- a view on features that answers the simple yet unanswered question as to why they are good and what do they represent,
- an interpretation that relates probabilistic (Monte Carlo-based) evaluation functions to deterministic features and describes, how both represent a slightly different solution to the same problem and how they can be combined.

Nevertheless, the focus of this work is on practical matters. Based on the above findings, our contributions are:

Construction of an Aggregation Function We propose a way to construct an aggregation function based on neural networks. The networks fuzzy-evaluate the game's goal condition on states and provide guidance as to how good states are for winning a game. Moreover, the resulting evaluation function needs no training, exhibits the properties of a high-quality evaluation function and allows for learning. As a side effect, an interesting solution for the integration of logic and learning systems is put into practice.

The approach will be discussed in section 3.2 and will be based on our publication *"Neural Networks for State Evaluation in General Game Playing"* at the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), 2009 [MT09].

Improved Aggregation Function based on Neural Networks The initialization algorithm ($C - IL^2P$) for the aggregation function is in some cases not suited for fuzzy-evaluating complex domains due to limited machine precision. We reduce the scope of this problem, thereby increasing the quality of the evaluation function.

The algorithm will be discussed in section 3.3, based on our workshop contribution *"Neural Networks for High-Resolution State Evaluation in General Game Playing"* to the International Workshop on General Intelligence in Game-Playing (GIGA), 2011 [Mic11b].

Rule-Derived Feature Acquisition We propose a general way to acquire features from formulas occurring in the game rules. The approach avoids expensive simulations for feature weighting or selection as used in other agents. The resulting feature acquisition provides useful features and is much faster than simulation-based approaches.

It will be discussed in sections 4.3 and 4.4 and is based on our workshop contribution *"Heuristic Interpretation of Predicate Logic Expressions in General Game Playing"* to the International Workshop on General Intelligence in Game-Playing (GIGA), 2011 [Mic11a].

Acquisition of Admissible Distance Features The distance features used to date are based on syntactic structures in the rules and cannot distinguish between move patterns of different objects. We address these problems for a class of distance features in section 4.5.

The underlying idea was first discussed in the workshop contribution *"Distance Features for General Game Playing"* to the International Workshop on General Intelligence in Game-Playing (GIGA), 2011 [Mic11a] and later published as *"Distance Features for General Game Playing"* at the International Conference on Agents and Artificial Intelligence (ICAART), 2012 [MS12]. Both articles are joint work with my former colleague and fellow PhD student, Stephan Schiffel.

The practical results will be summarized in the GGP agent Nexplayer.

1.3. Outline

In chapter 2 we will introduce traditional Game Playing and General Game Playing and concepts related with both areas. This will provide the necessary framework for the discussion and construction of an aggregation function in chapter 3. In chapter 4 we will proceed by discussing features and how they can be integrated in our aggregation function. We will combine both, aggregation function and features, in our agent Nexplayer and evaluate it in chapter 5. Finally, we will discuss related approaches in chapter 6 and conclude with a summary and future work in chapter 7.

2. Game Playing

Game Playing is interesting for research in Artificial Intelligence because games have various qualities

- Games are problem domains of small size
- Games can be easily explained and communicated to other people
- Games have a recreational annotation
- Games encourage competition and, thereby, evaluation
- Games are complex systems, i.e., seemingly simple decisions may have huge consequences

Specifically the latter point also explains the economic motivation behind research in Game Playing: Agents that master complex domains should also be able to perform other tasks that require intelligence. The ultimate goal is to substitute human labor by cheap agent labor. Producing an agent that matches the playing strength of a human is a first step on this road.

One of the earliest works on the topic of game playing was released in 1950 by Claude Shannon [Sha50], however it should take another 40 years before algorithms and hardware were capable of matching the strength of a human. The following is a list of milestones where after a long period of struggle between humans and intelligent agents the latter emerged victorious.

Backgammon TD-Gammon [Tes95] in addition to its strength used tactics that were later on widely adopted by human players

Chess Kasparov's defeat in Chess against Deep Blue in 1997 [CJhH02]

Checkers the complete solution of the game Checkers by Schaeffer [SBB+05], which since then is known to end in a draw if both sides play optimally

The aforementioned successful agent programs have in common that the vast amount of work is done by their developers: They define the agent characteristics, such as search strategy, state evaluation functions and data representation, such that they fit to the specific game the agent is designed for. Chess agents are designed with the help of grandmasters that select and weight features for the evaluation function, TD-Gammon has a state representation especially designed for backgammon and the Checkers engine Chinook plays optimal because an endgame database was grown over years such that today it covers the initial states.

So instead of showing a flexible and intelligent behavior, the agents reproduce the preprogrammed behavior of a presumably intelligent developer.

To overcome this problem, research in General Game Playing was proposed[GLP05] and eventually emerged over the last few years. The idea is that by decoupling agent programming and the game, programmers cannot adapt their agents anymore to a specific game but have to endow their agents with autonomous planning, learning and reasoning facilities to handle a game. From the perspective of an agent, a typical GGP match starts thus with the reception of the game rules and a predefined amount of time until the first move has to be played, followed by a regular submission of its moves until a terminal state is reached.

This chapter serves as an introduction to Game Playing and General Game Playing. We start by defining general games in the next section. We then discuss the most important notions needed for understanding Game Playing in section 2.2. We proceed in section 2.3 by discussing the move selection problem that an agent faces and how evaluation functions can be used to address the problem to some extent. Search algorithms complement this solution and are presented in section 2.4. Finally, we discuss how to theoretically (section 2.5) and practically (section 2.6) assess evaluation functions.

2.1. General Game Playing

General Game Playing is the next logical step on the way from traditional Game Playing towards producing intelligence in agents. It effectively prevents the high-specialization approaches of traditional game playing by distributing the game rules at run time. Thus, data structures cannot be optimized for a specific game and humans cannot provide game-specific heuristics to guide search.

The insights provided by GGP are as applicable as in traditional Game Playing but much more general. Any problem description representable as a game can be put for analysis by a GGP agent. The agent will play and thereby represent a strategy on how to act in a situation, effectively offering decision support. Moreover, if the problem description changes, the agent does not need to be modified at all.

In addition to these agent-centric applications, GGP may be used to model complete environments. The basic rules of the environment and a set of agents are all that is needed. Practically all reward-based real world systems are instances of such an environment model with the best known being markets and evolution. However, practical applications such as modeling political decision processes can be derived straighforwardly.

Finally, in the thesis we argue that strong (i.e., well playing) agents need to exhibit properties of a prediction system for terminal states. The games used in GGP can thus be seen as an instance of other prediction systems with the basic difference being an accessible formal description of the goal and environment dynamics.

In order to describe general games, the Game Description Language (GDL) was defined [LHH$^+$08]. An additionally defined communication protocol based on HTTP ensured the compatibility of all agents via a common interface. Each year a competition

is held where agents demonstrate the state of the art.

In this section we will first informally describe what a game is. We proceed by showing how this is expressed in GDL, followed by a formal semantic of a game description in GDL. Finally, we briefly explain the communication protocol.

Note that there is an extension to GDL, called GDL-II, that enables the encoding of games with incomplete and/or imperfect information and non-determinism [Thi10]. However, this work will based solely on the original version of GDL.

2.1.1. GDL Described Informally

A GDL game is played by players that occupy roles one-to-one. Such a role could be "white" in Chess or "attacker" in a war game. These players start a match in the initial state and choose each a legal action (also called move) for their role. After submitting the action, the state changes and each player again chooses and submits an action. The process is repeated until a terminal state is reached. Each role is then assigned points in accordance with its goal.

A GDL game is thus defined by

- roles
- initial state
- legal moves (depending on role and current state)
- state update function (depending on current state and moves selected)
- terminal function (returns true/false depending on the current state)
- goal function (returns a numeric value depending on the role and current state)

2.1.2. Game Syntax: Game Description Language

The rules of a game are represented as a logic program [Llo87], including negation. We assume familiarity with the basic notions of logic programming.

Definition 2.1 (Logic Program)**.**

Term A term is a variable or a function symbol applied to terms as arguments. A constant is a function symbol with no argument.

Atom An atom is a predicate symbol applied to terms as arguments.

Literal A literal is an atom or its negation (indicated by $\neg$).

Clause A clause is an implication of the form $h \Leftarrow b_1 \wedge \ldots \wedge b_n$ where h is an atom and the body $b_1 \wedge \ldots \wedge b_n$ is a conjunction of literals b_i with $0 \leq i \leq n$.

Logic Program A logic program is a set of clauses.

We will refer sometimes to a function or predicate symbol p with arity n as p/n. n here is the arity of the corresponding predicate or function and indicates the number of arguments it takes, i.e., p/n refers to a predicate p of the form $p(x_1, \ldots, x_n)$.

Also, we define a formula as either a literal or, recursively, as a conjunction or disjunction of two formulas.

The Game Description Language (GDL) encodes a game (as informally described before) as a logic program and happened to become the standard language for encoding game rules of general games.

While GDL allows for deviations from the syntax of a logic program (specifically, formulas in the body of a clause), we will assume adherence to the logic program syntax. This promotes simplicity and does not restrict the expressiveness of GDL since the clauses in question can be easily transformed to logic program clauses.

GDL uses specific predicates called keywords that represent game-specific attributes. We will give GDL notations in an easy-to-read Prolog-style where variables are denoted by uppercase letters and predicates and functions by lowercase letters. The symbols $\Leftarrow$ and $\wedge$ are replaced by the symbols `:-` and `,` and each clause is terminated with the symbol `.` (a dot). A clause with empty body is represented as a fact, e.g., $h \Leftarrow$ is represented as `h.`

The initial state is defined by the clauses with the keyword **`init/1`**. Roles are defined using the keyword **`role/1`** and given as ground assertions. Both keywords describe essential elements of the game and are independent from the current state.

Keywords	Explanation
`role``(R).`	`R` is a role in the game
`init``(P) :-...`	`P` is a fact holding in the initial state

Table 2.1.: State-Independent GDL Keywords

For each state, it can be determined whether the state is terminal using the **`terminal/0`** keyword and in this case a goal value for each role can be obtained using the **`goal/2`** keyword.

If the current state is non-terminal, then the **`legal/2`** relation defines for each role the set of legal moves. Each agent submits a legal move and the new state can be determined using the **`next/1`** keyword. The current state is then discarded and the new current state is defined as the facts entailed by **`next/1`**.

Keywords	Explanation
`legal``(R, A) :-...`	`A` is a legal action for role `R` if ...
`next``(P) :-...`	`P` holds in the next state if ...
`terminal``:- ...`	the current state is terminal if ...
`goal``(R, V) :-...`	role `R` achieves a goal value of `V` if ...

Table 2.2.: GDL Keywords allowed only as rule heads

Each of these predicates may depend on the current state and the player's actions. Such conditions are formulated using the following keywords:

Keywords	Explanation
`does(R, A)`	role `R` does action `A`
`true(P)`	`P` holds in the current state
`distinct(X, Y)`	`X` does not unify with `Y`

Table 2.3.: GDL Keywords allowed only in rule bodies

Besides, there are other restrictions for valid GDL [LHH+08], however, these are mostly intuitive and will be mentioned when necessary.

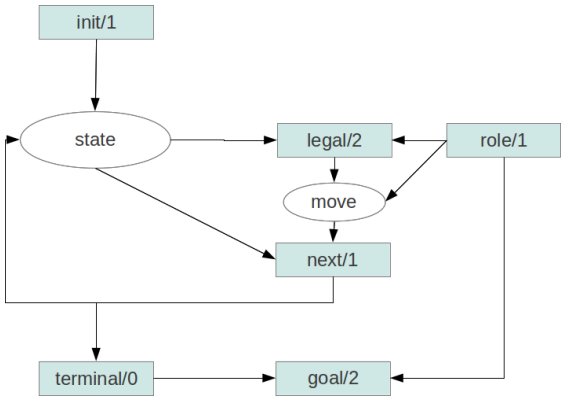


Figure 2.1.: Data Flow Within an Agent

Figure 2.1 shows the data flow in a GGP environment. All colored boxes represent GDL relations and all white ellipses represent variable sets of facts that are defined only implicitly. An arrow indicates that the information encoded in the arrow source is required for the deduction of the arrow target. The variable sets representing the current state and the selected moves are queried using `true/1` and `does/2` statements, respectively.

2.1.3. GDL rules of Tic-Tac-Toe

Here and in the remainder of this work we will refer to a number of games in order to provide an example. Each of these games is accessible on the GGP Server [Sch12].

The following is an encoding of a Tic-Tac-Toe game in GDL that illustrates how a game is described in GDL. Keywords are printed in bold, comments start with a percentage symbol `%` and are printed italic.

```
1 role(xplayer). % defines the roles xplayer ...
2 role(oplayer). % and oplayer
3
4
5
6 % sets all cells to b (blank)
7 init(cell(1,1,b)). init(cell(1,2,b)). init(cell(1,3,b)).
8 init(cell(2,1,b)). init(cell(2,2,b)). init(cell(2,3,b)).
```

```
9  init(cell(3,1,b)). init(cell(3,2,b)). init(cell(3,3,b)).
10 % sets the control toggle to xplayer
11 init(control(xplayer)).
12
13 % P has the legal move ...
14 legal(P, mark(X, Y)) :- % ... mark(X, Y)
15   true(control(P)),     % if P has control
16   true(cell(X, Y, b)).  % and cell (X,Y)  is blank
17
18 legal(P,noop) :-        % ... noop
19   role(P),              % if it does not have control
20   not true(control(P)).
21
22 % switches the control every move
23 next(control(xplayer)) :- true(control(oplayer)).
24 next(control(oplayer)) :- true(control(xplayer)).
25
26 % in the next state cell(X,Y) contains ...
27 next(cell(X,Y,x)) :-        % ... an x
28   does(xplayer,mark(X,Y)).  % if xplayer moves 'mark(X,Y)'
29
30 next(cell(X,Y,o)) :-        % ... an o
31   does(oplayer,mark(X,Y)).  % if oplayer moves 'mark(X,Y)'
32
33 next(cell(X,Y,C)) :-        % ... content C
34    true(cell(X,Y,C)),       % if it already contains C ...
35    distinct(C, b).          % and C is not blank
36
37 next(cell(X,Y,b)) :-        % ... blank
38   true(cell(X,Y,b)),        % if it is blank
39   does(P, mark(M, N)),      % and P marks a different cell
40   (distinct(X, M) ; distinct(Y, N)).
41
42 % xplayer wins ...
43 goal(xplayer, 100) :- % ...100 points
44   line(x).            % if there is a line of x
45
46 goal(xplayer, 0) :-   % ... 0 points
47   line(o).            % if oplayer achieves a line of o
48
49
50 goal(xplayer, 50) :-  % ... 50 points
51   not line(x),        % if there is no line of x
52   not line(o),        % no line of o
53   not open.           % and the board is not open
54
55  % same for oplayer
56 goal(oplayer, 100) :- line(o).
57 goal(oplayer, 50) :- not line(x), not line(o), not open.
```

```
goal(oplayer, 0) :- line(x).

% the game ends if ...,
terminal :- line(x).  % ...there is a line of xs
terminal :- line(o).  % ...there is a line of os
terminal :- not open. % ... or the board is not open.

% a line of C consists of
line(C) :- row(X, C), index(X).   % ... a row of C
line(C) :- column(Y, C), index(Y).  % ... a column of C
line(C) :- diagonal(C).       % ... or a diagonal of C

row(X, C) :-            % a row of C is ...
  true(cell(X, 1, C)),  % cells (X,1)
  true(cell(X, 2, C)),  % (X,2) and
  true(cell(X, 3, C)).  % (X,3) having the content C

column(Y, C) :-         % similar for columns
  true(cell(1, Y, C)),
  true(cell(2, Y, C)),
  true(cell(3, Y, C)).

diagonal(C) :-          % similar for upward diagonal
  true(cell(1, 1, C)),
  true(cell(2, 2, C)),
  true(cell(3, 3, C)).

diagonal(C) :-          % similar for downward diagonal
  true(cell(3, 1, C)),
  true(cell(2, 2, C)),
  true(cell(1, 3, C)).

% the board is open if there is a blank cell.
open :- true(cell(X, Y, b)).
```

2.1.4. Game Semantics

A formal semantic that interprets a game description D in GDL as a state transition system was first provided by [ST10]. It contains all elements as given in the prior informal description and is obtained by treating the game description D as a logic program. In order to represent the current state and actions selected by the players, we additionally define the s^{true} and A^{does} as ground logic programs.

$$s^{\texttt{true}} \stackrel{\text{def}}{=} \left\{ \begin{array}{c} \texttt{true}(p_1) \\ \ldots \\ \texttt{true}(p_n) \end{array} \right\}$$

$$A^{\texttt{does}} \stackrel{\text{def}}{=} \left\{ \begin{array}{c} \texttt{does}(r_1, A(r_1)) \\ \ldots \\ \texttt{does}(r_n, A(r_n)) \end{array} \right\}$$

$s^{\texttt{true}}$ here is a representation of a state. Each element $p_1, \ldots, p_n$ is a fluent, that is, a term that occurs as argument of the keywords **`next`**, **`init`** or **`true`**.

$A^{\texttt{does}}$ is a representation of a joint action or joint move, that is, the set of moves selected by each player.

The game Γ is then defined in terms of the game description D and the descriptions of $s^{\texttt{true}}$ and $A^{\texttt{does}}$:

Definition 2.2 (Game Semantics)**.** Let D be a valid GDL specification, whose signature determines the set of ground terms Σ and 2^{Σ} denotes the set of finite subsets of Σ. The semantic of D is a game $\Gamma = (R, s_0, T, l, u, g)$ where

- $R = \{r \in \Sigma : D \models \texttt{role}(r)\}$;
- $s_0 = \{p \in \Sigma : D \models \texttt{init}(p)\}$;
- $T = \{s \in 2^{\Sigma} : D \cup s^{\texttt{true}} \models \texttt{terminal}\}$;
- $l = \{(r, a, s) : D \cup s^{\texttt{true}} \models \texttt{legal}(r, a, s)\}$ where $r \in \Sigma, a \in \Sigma, s \in 2^{\Sigma}$;
- $u(A, s) = \{p \in \Sigma : D \cup s^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{next}(p)\}$ for all $A : (R \mapsto \Sigma)$ and $s \in 2^{\Sigma}$;
- $g(r, n, s) : D \cup s^{\texttt{true}} \models \texttt{goal}(r, n)$ where $r \in R, n \in \mathbb{N}, s \in 2^{\Sigma}$.

During the remainder of this work we will stick to the notation as outlined in this definition. Most important among these are the symbols

- $s \in \mathcal{S}$ to refer to a state
- $t \in T$ to refer to a terminal state
- $r \in R$ to refer to a role and
- $g(r, n, s)$ to refer to the goal relation

Given a game Γ with goal relation $g(r, n, s)$ we additionally define:

Definition 2.3 (Goal Function)**.**

$$\mathit{goal}(s, r) \stackrel{\text{def}}{=} n \text{ such that } g(r, n, s)$$

In addition, we will use the macro *holds* that evaluates a GDL formula in a state and is true if and only if the formula holds in the state.

$$\mathit{holds}(\epsilon, s) \stackrel{\text{def}}{=} D \cup s^{\texttt{true}} \models \epsilon$$

2.1.5. Game Manager

The game manager is the central entity that coordinates the agent programs. Initialized with the game rules, it sends a game start message plus the game rules to all participating agents. This message additionally includes

- the role the agent occupies in the game (e.g., whether the agent plays white or black)
- the start clock parameter indicating the number of seconds until the match starts and
- the play clock parameter indicating the time window (number of seconds) for submitting moves for each new state.

Each play message includes the moves submitted by every agent in the last step and every agent has to submit an action allowed by the game rules within the give time. If for some reason an action is submitted too late, a legal action will be assigned to the player. After the expiration of the play clock the game manager informs each agent of the moves selected. This process continues until a terminal state is reached. For terminal states, a game stop message is sent.

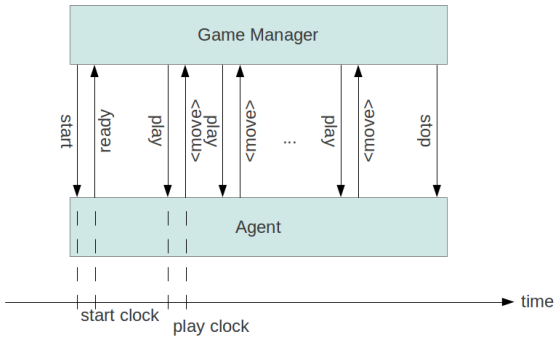


Figure 2.2.: Control Flow between Agent and Game Manager

Figure 2.2 shows the basic control flow between an agent and the game manager and illustrates the effect of the start clock and play clock parameters.

For a more detailed description see [LHH+08].

2.1.6. GGP Competitions

There is an active research community that evaluates new approaches in regular tournaments. The main event is an annual competition organized by the Stanford University[1], usually held together with the IJCAI/AAAI conferences. Also, in 2011 the "German Open in General Game Playing" as another tournament with international participation

[1] games.stanford.edu

were first held, organized by the Bremen University[2]. To our knowledge there are also several minor tournaments among students of the universities of Dresden, Potsdam and Reykjavik.

The setup of a typical tournament consists of matches in a number of newly defined games with a start clock of up to a few minutes and a play clock of up to one minute.

[2] http://www.tzi.de/~kissmann/ggp/go-ggp/

2.2. Basic Notions of Games

Before delving into the matter of evaluation functions, we briefly review some basic notions that we deem indispensable in the discussion of Game Playing. These are presented next.

2.2.1. Dimensions of Games

General Game Playing deals with deterministic synchronous games with complete information. This class of games is considered interesting for research because it represents the simplest class of problems still hard to solve.

Deterministic A game is deterministic if in any state for any legal joint action there is exactly one successor state. Games that include random elements such as rolling a die are not deterministic.

Complete Information A game is a complete information game if the rules of the game are known to every player and at every point of time during a match the complete state information is fully and correctly perceivable by all players. The opposite are incomplete information games where state (or rule) information is hidden, e.g., by cards visible only to some players.

Synchronous A game is synchronous if each player may submit only a single action and the actions of each player become effective only at a predefined point of time. Consequently, while no move is made, the game remains in a state that persists until the next joint action enters into effect. The opposite of synchronous games are real-time games where players are free to act, how often and when they want. Consequently, the state of a real-time game can change at any given time.

For handling games, we will additionally make use of the following properties that divide the class of GDL games into further subclasses.

Sequential A subclass of synchronous games are sequential games. A game is sequential if in each state exactly one player is allowed to act. The opposite is a simultaneous game where in at least one state players act at the same point of time. Both dimensions are also referred to as alternating move vs. simultaneous move games. Sequential games are often modeled as simultaneous games by assigning dummy (or noop) moves to all players that in the sequential version would be not allowed to move. These noop moves have no effect on the game state.

Zero-Sum A game is a zero-sum game if for each terminal state the sum of the rewards of all roles is zero. Any game that produces a constant sum of rewards can be formulated as a zero-sum game, therefore we will use the two terms interchangeably. The opposite of a zero-sum game is a non-zero-sum game. Here players can achieve outcomes that are independent of each other.

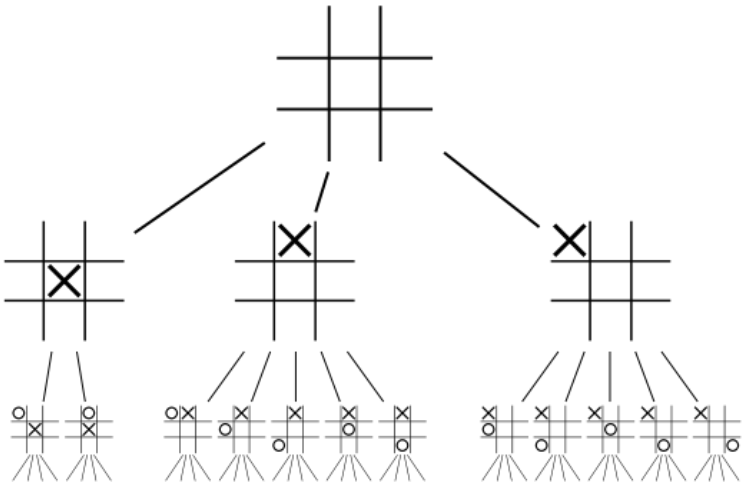


Figure 2.3.: A Game Tree for Tic-Tac-Toe

Besides, it may be worthwhile to mention that GDL games are Markovian, meaning that successor states depend only on the current state and the joint actions taken. Past states have no influence whatsoever, thus simplifying the problem.

2.2.2. Game Tree

We can depict the state space of a GDL game as a directed tree. Each node in the game tree is a state. The root is the initial state s_0 and all leafs are terminal states. There is a connection from state s_1 to s_2 if there is a legal joint move such that $u(A, s_1) = s_2$.

Technically, this definition implies that the game tree is a rooted directed acyclic graph since in some games the same state can be reached via different paths. However, we stick to the common term game tree.

A match is a path through the game tree from the root to a leaf. The aim of each player is to influence this path via its contribution to each joint action A such that his associated role achieves a maximum goal value.

Figure 2.3 shows a game tree for Tic-Tac-Toe. The root node is the initial state s_0 and is connected to all states reachable from the initial state. For the sake of simplicity we omitted all symmetric states.

2.3. Move Selection and State Evaluation

When an agent participates in a match as role r, he can be formally seen as implementing the function $\Pi_r(s)$ that maps a state s to a legal action $\Pi_r(s) \in L(r, s)$ where $L(r, s) \stackrel{\text{def}}{=} \{a : l(r, a, s)\}$ is the set of legal actions of role r in state s.

This function is called policy and fully determines the behavior of the agent.

$$\Pi_r(s) = a \text{ with } a \in L(r, s)$$

The policy can be thought of as a lookup table. Two agents with the same policy are indistinguishable from the outside since their behavior is equal.

Although the agent is allowed to submit only one action in a state, it may identify several actions as suitable. In this case the policy must be described non-deterministically and represented as a mapping of a state-action-pair to a probability $\Pi_r(s, a) = p$. Naturally, for each state the sum of probabilities for each legal action must be 1 and each single probability for any state-action-pair must be non-negative.

$$\begin{aligned} \Pi_r(s, a) &\geq 0 \\ \sum_{a' \in L(r,s)} \Pi_r(s, a') &= 1 \end{aligned}$$

The aim of agent programming is to find a policy that obtains the maximum possible reward for the role of the agent. This policy is called *optimal policy*. Correspondingly, for games with unforeseeable game elements such as for multiplayer games, nondeterministic games and games with only partially observable states expected (a priori) or average (a posteriori) reward must be maximized.

To obtain such an optimal policy the agent needs to decide in a specific state which of all legal actions is best. Hence a measure of preference of actions is needed. An analysis of the plain action is only possible if a context (i.e., the game) provides some sort of utility for an action. However, in most games, including GDL games, there is no such context as actions have no meaning other than being the vehicle to induce a state transition. Therefore, action consequences, i.e., successor states, are used as primary mean for comparing actions. Two actions can hence be compared by comparing the states they lead to. This process is called a single-ply look-ahead.

Unfortunately there is no easy way to compare states. Ideally, an agent would have a state value oracle that maps states to values and indicates the best state by assigning the highest value to it. However, since the value of a state is determined by the value of the terminal state reached given optimal play. Thus, a practical implementation of a state value oracle requires a move selection oracle, bringing us back to our first problem.

Still, all is not lost since by reducing move selection to state evaluation and back to move selection, we effectively advance on level in the game tree. By applying this step repeatedly, we have the foundation for search algorithms that can be used to address the problem (see section 2.4). At the same time, we can try to approximate the state value oracle using an own evaluation function. In practice, both, search and evaluation functions, are combined to address the move selection problem.

In the remainder of this work, we will refer to the state value oracle as perfect evaluation function and mean a function that evaluates the n^{th} best move to the n^{th} highest state value.

2.3.1. Deterministic Evaluation

Deterministic evaluation functions derive their name from the fact that they map the same state to the same value, like the goal function of a game. Since the goal function is often not sufficient to distinguish non-terminal states from each other, other functions are used that indicate the value of a state. Deterministic evaluation functions employ for this purpose subfunctions, called features, that evaluate the degree of presence or absence of concepts associated with winning the game. The evaluations of several features are then aggregated and transformed to a single value, the state value.

An example for such a function is a Chess evaluation function: There are many concepts such as king safety, pawn structure and piece value. Each of these may be represented as a feature. For example, the piece value feature counts the number of pieces times their corresponding piece value and returns an evaluation that indicates the strength of the pieces in a state. The feature reflects the chess rule of thumb that the more pieces you have and the more powerful they are, the higher are the chances of winning.

An evaluation function based on the features piece value and king safety would determine the corresponding feature evaluation in a state and then aggregate them to a single state value. Using this function for evaluating moves thus would guide an agent towards maintaining and increasing the value of its pieces while keeping the king safe, both of which is a good heuristics for Chess.

A closely related dimension of evaluation functions is the class of knowledge-based evaluation functions since deterministic functions typically involve the discovery and use of features based on domain knowledge. In fact, it makes most sense to deterministically evaluate a state if you have knowledge about what the state value could be. However, both classes are not the same since deterministic functions can be knowledge-free and knowledge-based functions are not required to be deterministic.

Aggregation Function and Features

It is common to construct a deterministic evaluation function as a function with nested functions as arguments.

The nested functions with no functions as arguments are called features. They operate directly on the state and return a numerical value. All other functions are called aggregation functions since they aggregate the numerical values of other functions to a single value.

In figure 2.4 the hierarchy of functions is shown as a tree, features are leafs and aggregation functions are inner nodes of the tree.

The most popular example of a deterministic evaluation function is a linear combination of features, as given in figure 2.5. Here, the feature output values for a state are aggregated by calculating their weighted sum. The weight of each feature is set to correspond to its quantified importance.

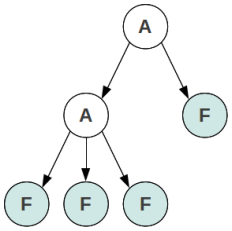


Figure 2.4.: Example for an Evaluation Function consisting of Features (F) and Aggregation Functions (A)

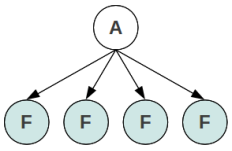


Figure 2.5.: A Linear Combination of Features

2.3.2. Probabilistic Evaluation

A probabilistic evaluation function determines the value of a state as the value of a terminal state given random play from that state. Technically, this means that a probabilistic evaluation is not a function, however, for our purposes it is convenient to treat it as such.

Given a state s, a probabilistic evaluation is realized as a simulation, where a random joint action is performed in s yielding a successor state s'. In this state again a random action is performed and the process is repeated until a terminal state t is reached. A sequence of states generated by random actions starting in s and ending in a terminal state t is called a (random) *playout*. The value defined by the goal function on t is the corresponding playout value and the return value of the probabilistic evaluation function applied on s.

Since the result of a probabilistic evaluation is random, the search algorithm UCT (Upper Confidence Bounds applied to Trees), discussed in section 2.4.3, is typically used in combination with probabilistic evaluation because it repeats playouts and uses the average playout value for guidance.

2.3.3. Comparison: Probabilistic vs. Deterministic

There is a great deal of uncertainty whether probabilistic or deterministic evaluation functions are the better choice for state evaluation. While deterministic evaluation functions work well in many games such as Chess, all GGP competitions since 2009 were won by agents with probabilistic evaluation functions. Before giving our own opinion on the subject, we present some of the most important arguments in favor and against each. We opt to discuss only the properties of probabilistic evaluation. For any of their

advantages/disadvantages discussed, it should be easy to identify the corresponding disadvantage/advantage of a deterministic evaluation. We begin with the advantages.

First, probabilistic functions only need a generative model of the environment and a goal or reward function. Based on this, they can randomly generate actions, execute them and eventually obtain a reward. Therefore there is no need to create an explicit evaluation function, the simple model-based approach works as an *implicit evaluation* and always provides results. For the same reason, the evaluation function can be *developed easily*.

Second, probabilistic evaluation functions are very *flexible with respect to the requirements to the environment*. The generative model needs not be explicit. Instead, black-box functions for action generation, action execution, state update and goal assignment are sufficient. This implies that the rules or the game state could be only partially observable while the system would still work.

Third, from an empirical point of view there are games where no evaluation function could be found up to now. The outstanding example is Go [GWMT06, GS07] where states are believed to be too unstable to represent a stationary situation useful for evaluation since the presence or absence of a single stone can turn the evaluation completely. A probabilistic evaluation seems to be the only viable function for *games with unstable states*.

On the other hand, probabilistic evaluation also brings a number of disadvantages: The major disadvantage is the probabilism itself, since it assumes that *all players act randomly*. This assumption renders the evaluation bad against experts.

Besides, there are game-specific factors that may put a probabilistic evaluation function at a disadvantage, namely a big game tree to traverse and non-descriptive terminal states. Particularly, Chess exhibits characteristics that illustrate the shortcomings of a probabilistic evaluation:

High Branching Factor For most of the game the average number of moves is around 30 [Sha50].

High Depth A Chess match can potentially run for a long time, i.e., there is no predefined number of steps after which the match stops.

Low Probability for Descriptive Terminal States When, eventually, a terminal state is found, it is not necessarily descriptive, i.e., it is most likely terminal state is a draw due to repetition or the 50 move rule.

Game-specific disadvantages are mitigated by the UCT search algorithm (see next section) typically applied together with probabilistic evaluation once the (possibly erroneously) evaluated state enters the in-memory tree. However, for this to happen, the probabilistic evaluation has to be descriptive to some extent, and the in-memory tree should contain preferably many successor states of the erroneously evaluated state. The latter, again, is difficult, given that the overhead of UCT grows with the depth of the in-memory tree, as, unlike in iterative deepening, the complete path from the root to a leaf has to be traversed again for each single state evaluation.

Finally, the main advantage of an implicit evaluation is also its greatest disadvantage because rule-inherent abstract knowledge is ignored.

2.4. Search Algorithms

Since we cannot assume an evaluation function of an agent to be optimal, we must consider it an approximation of an optimal evaluation function. Therefore we look for techniques to minimize the approximation error and the most straightforward way to do this is search.

Recall that move selection can be reduced to evaluating the successor states of each move. We can, for example, apply this technique to find the best move in states of single-player games that are direct predecessors of terminal states, since in this case the goal function returns optimal state values.

This approach can be generalized to holding for any predecessor state of a terminal state by viewing the problem as a game-theoretic one: Each player has a finite amount of actions and for each possible joint action the payoff is defined by the goal function applied on the resulting successor state. Hence we are able to depict the move selection problem as a simple one-step normal-form game which can be represented by a tensor of dimension $|R|$ (the number of roles) with each role's number of legal actions $|L(r, s)|$ defining the number of entries of the respective dimension. If there is a single (possibly mixed-strategy) Nash equilibrium we can calculate the average outcome per player in that state and hence know the state value in that state.

Applying this algorithm backwards starting from the predecessors of terminal states towards the initial state, we can determine the exact value of all states and therefore also find the optimal evaluation function and policy.

This means generally that we can derive the exact value of a state if we know the exact values of its successors.

However, there is a number of problems related to the approach.

First, the problem is recursive since we try to obtain a state's exact value by aggregating its successor states' exact value. However, only in very few cases we have the exact state values. The problem is typically addressed by substituting the exact values by an approximation of the exact values, i.e., their values as determined by our evaluation function. Though this may seem counterintuitive, the aggregated state evaluations of successor states can be expect to be better approximations than the simple evaluated state because the successor states inspected are closer to the terminal state that will eventually be reached.

The second problem is that there may be more than one Nash Equilibrium. In this case, there is no optimal solution and one is forced to choose one of the equilibria, be it randomly or based on other criteria such as opponent information or average points achieved per equilibrium.

The last problem is that the time needed for determining the Nash equilibria is exponential in the number of players. This may make finding the optimal state value practically infeasible. The problem is addressed in many ways. For example, the UCT

search algorithm sacrifices accuracy and relies on an approximation instead. Other search algorithms are based on simplifications that occur frequently in games such as:

1. If an action for a player returns a higher payoff than another action, regardless of the opponent actions, then the second action is said to be dominated and can be ignored.

2. If games are zero-sum, then the dimension of the payoff is reduced by one.

3. If a player in a game has only one legal move, such as in alternating move games, the dimension of the payoff tensor is reduced by one.

While each of these simplifications allows for general optimizations, specializations of these cases lead to well-known search algorithms: The search algorithm MaxN is used in games where only one player moves at a time and the search algorithm MiniMax is the two-player zero-sum specialization of MaxN.

In the following we will review the search algorithms MaxN, MiniMax and UCT.

2.4.1. MaxN

MaxN search is used in games with alternating moves. It assumes that each player plays for maximizing his own payoff. Since in each state only one player is allowed to move, the Nash equilibrium trivially leads to the state with the maximum reward to the acting player.

MaxN relies on the assumption that each agent is egoistic and plays for the maximization of his reward. This assumption, however, is not necessarily true, as another rational strategy for an agent would be to avoid moves where other players are awarded more points than the agent itself.

Due to potentially high resource requirements, a simple Breadth-First search is typically replaced by Iterative Deepening. Here, the tree is traversed in Depth-First order, but with an additional depth parameter that indicates the maximum depth of the search. If the state under inspection is at the maximum depth, then the state value is determined by the evaluation function, else the evaluation of the current state is defined as the (approximated) payoff of the payoff matrix implied by the values of the successor states.

2.4.2. MiniMax

MiniMax is a specialization of MaxN for two-player zero-sum games. Specifically, MiniMax does not need to distinguish between opponents maximizing their own score or minimizing the other player's scores as in zero-sum games this amounts to the same.

The name MiniMax is derived from the fact that in two-player games the aggregation of successor state values is a mere minimization if the opponent moves or a maximization if we move. Since rewards are tied to each other, various possible opponent strategies (such as maximizing the difference or ratio of values) all amount to the same. Hence there is only one opponent strategy and the value determined for the opponent is its

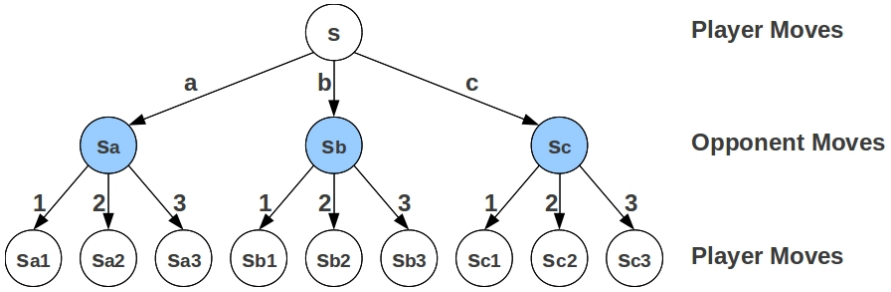


Figure 2.6.: States s_{b2} and s_{b3} can be pruned if $v(s_{b1}) < \min_x v(s_{ax})$

true value. In other words, the obtained state value corresponds to the state value in the Nash equilibrium.

Again, MiniMax is typically coupled with Iterative Deepening.

MiniMax can be generalized to simultaneous move games by following a paranoid approach: Though all players move at the same point of time, our player assumes that all opponents know his action before choosing theirs. In this way, MiniMax is worst-case optimal, however, this is not necessarily the best option and may lead to an underestimation of a state if players play for other objectives.

Alpha-Beta Pruning

It is common to extend plain MiniMax with the extension called alpha-beta pruning [Sch89].

Consider a state s to be evaluated, with the role r choosing between the legal actions $L(r, s) = \{a, b, c\}$. Each move leads to the successor states s_a, s_b, s_c, respectively, and role r is interested in maximizing the state value. In each of these successor states, the opponent has three moves $1, 2, 3$ that lead to the states $s_{a1}, s_{a2}, \ldots, s_{c3}$. Figure 2.6 illustrates the game tree.

Now assume that after the evaluation of all successors of s_a it was found $\alpha = \min_x v(s_{ax})$ where v is some deterministic evaluation function. If the player now finds another node s_{b1} with $v(s_{b1}) < \alpha$, the nodes s_{b2} and s_{b3} need not be examined, as it will not choose action b because the opponent would then choose action 1 in state s_b. Hence action b leads in any case to a value lower than action a and can be ignored. The nodes s_{b2} and s_{b3} can thus be pruned, speeding up the search.

Similarly, the opponent would not select the action 3 in s_c if he knew that the first action in s_{c3} would return a very high state value. Thus examining the remaining successors of s_{c3} can be skipped.

In summary, the window $[\alpha, \beta]$ represents the minimum and maximum value that the two players concede to each other. If a value outside of this window appears, the siblings of the corresponding state can be pruned.

In order to evaluate the minimum number of states possible, actions are typically ordered according to their perceived probability of allowing for pruning of other states.

The order is imposed by favoring high αs, low βs, or other heuristics (e.g., history or killer heuristics [Sch89] that promote moves that produced cut-offs earlier).

2.4.3. UCT

UCT (Upper Confidence Bounds applied to Trees [KS06]) represents an optimal solution to the exploration-exploitation dilemma in Markov Decision Processes and recently became popular due to its striking simplicity and ease of parallelization.

Consider a single-player game. A UCT exploration episode takes a state s, updates an in-memory representation of the state space explored from s and returns a value for the s. It consists of four steps:

Selection Step Starting from s, the existing in-memory game tree (UCT tree) is traversed and in each state an action is selected according to equation (2.4.1).

Expansion Step Once an action with no information attached is found, the action is selected and the resulting state added to the in-memory tree.

Evaluation Step The state is evaluated using the evaluation function.

Backpropagation Step The state value determined is backpropagated to all predecessor states that were passed during the episode.

The key in the algorithm is the selection step that directs search through the in-memory part of the game tree. Note that this is different from the action selection by an agent in a match since the actions used for exploration are only executed in memory and have no effect on the environment. The final move of an agent is still selected greedily among all actions.

For each state s that is in the UCT tree, an action a is selected the following way:

$$\Pi_{UCT}(s) = \arg\max_{a \in A} \left(Q(s,a) + C\sqrt{\frac{lnN(s)}{N(s,a)}} \right) \tag{2.4.1}$$

The action selected depends on two terms: The first term $Q(s,a)$ is the average value associated to the state-action pair (s,a) as used in the Monte-Carlo estimation, that is, the average of all episode goal values whose episodes went through s and selected a.

The other term $C\sqrt{\frac{lnN(s)}{N(s,a)}}$ is the UCT bonus with $N(s)$ being the number of visits to state s and $N(s,a)$ the number of times a was selected as action when an episode traversed s. C is a parameter that determines the influence of the UCT bonus.

The UCT bonus for any pair (s,a) increases if the state s is visited without having chosen a as action. On the other hand it decreases, when (s,a) was selected. Thus with each episode the game tree grows asymmetrically into the direction of the most promising states while remaining shallow in suboptimal ones. Nevertheless every action is explored once and even the worst action will be explored further at some point of time.

In n-player games for $n > 1$, UCT can be applied by modeling the game as a set of n single-player games in a non-deterministic environment. Each role is unaware of the presence of other roles and the corresponding agent simply chooses one of its legal actions. The choices of other agents are hidden and are modeled as behavior of the non-deterministic environment. The only connection between the agents are environmental responses and the eventual reward. Consequently, each role has its own tree storing the values $Q(s,a), N(s), N(s,a)$ and chooses its action independently. Thus, a deterministic successor state cannot be determined and reasoning over states (instead of actions) is not possible.

2.5. Criteria of Evaluation Functions

In order to construct an own evaluation function, we must be aware of the properties that render an evaluation function good. Good here means a high expected probability that the moves implied by the evaluation function lead to terminal states with a high value, regardless of the opponent. Since the values determined by evaluation functions cannot be considered accurate, they are usually combined with search to mitigate eventual approximation errors. Thus efficiency comes into play as more efficient functions allow for the inspection of a higher number of states in a given amount of time.

In practice, efficiency requires the evaluation function to adhere to time and space constraints. Specifically, both, the space occupied by the function representation and the time required to perform an evaluation must be bounded. Simplistic evaluation functions such as exhaustive search or lookup tables with time or space requirements linear in the size of the state space can thus be quickly dismissed.

In addition to these criteria occurring in classic game playing, GGP requires a fast construction of the evaluation function. Consequently, plain learning approaches cannot be found among the GGP agents that participate in the competitions due to their time-consuming nature.

Quality requirements are more difficult to analyze since they are hard to quantify. In the following we assume a perfect evaluation function $v^*(s)$ as a function that evaluates all states to the value they eventually yield given optimal play. This definition coincides with the state value oracle assumed in section 2.3.

The perfect evaluation function is also optimal, i.e., it is a move selection oracle, since it always evaluates the moves that lead to the best successor states highest. However, in contrast to an optimal function, it also evaluates n^{th} best move to the n^{th} highest value. This property is useful since our (approximated) evaluation function cannot be considered optimal, i.e., it does not assign the highest value to the best state, and therefore benefits from correctly identifying the second-best (and third-best, ...) state among the successor states.

We use the perfect evaluation function to define properties of our evaluation function relative to that of the perfect evaluation function. While the list of properties is not complete, it will be sufficient to illustrate the individual strengths and weaknesses of each GGP agent system. By requiring that our evaluation function $v(s)$ be an approximation

Efficiency	Quality
Compactness	Consistency
Bounded Computation Costs	Strict Monotonicity
Ad-Hoc Construction	Extrapolation

Table 2.4.: Summary of the Evaluation Function Criteria

of $v^*(s)$ we can conclude the following properties:

Consistency $v_{(1)}(s,r) = v_{(2)}(s,r)$

The equation states that the evaluation of a state is independent of any circumstances represented by the symbols (1) and (2). In other words, the value of a state depends only on the state and the role. Consistency is useful since inconsistent evaluations imply an estimation error. This error can be reduced by repeated evaluations to the detriment of computation costs. However, the negative impact of inconsistent evaluations cannot be completely removed.

Strict Monotonicity $\forall s_1, s_2 \in \mathcal{S} : v^*(s_1,r) > v^*(s_2,r) \Leftrightarrow v(s_1,r) > v(s_2,r)$

Strict monotonicity ensures that we are able to rank states according to their perfect state value and that differences in the perfect state values $v^*(s,r)$ are preserved by our evaluation function. This implies that symmetric states are assigned the same values.

While strict monotonicity is hard to measure given that we do not know the perfect evaluation function, we can construct cases where we know that a given state should be evaluated higher than another. The evaluation function then should reflect this valuation.

Extrapolation $\forall s \in T : goal(s,r) = v(s,r)$

Since $v^*(s,r)$ coincides with the goal function in the terminal states, our evaluation function $v(s,r)$ must approximate the goal function in the terminal states, that is, $v(s,r)$ should be an extrapolation of the goal function (from T to $\mathcal{S}$). Although this property can be synthetically derived by overriding any evaluation function in the terminal states with the value obtained by the goal function, the resulting function most likely exhibits a discontinuous behavior, misaligning the state values of terminal versus that of non-terminal states.

Note that each of the criteria is a heuristic, meaning that it should be possible to create an optimal evaluation function that exhibits none of the aforementioned properties. However, given the difficulty of finding an optimal evaluation function, we believe chances of success are much higher when accepting the properties as guidance.

Table 2.4 summarizes the criteria.

2.6. A Word on Experimental Evaluation

In order to verify the ideas presented in this work, we will conclude each practical part with an experimental evaluation. Since an evaluation function itself is difficult to assess, we integrate the evaluation function in our agent Nexplayer. Consequently, we can assess the quality of an evaluation function by evaluating the quality of the agent Nexplayer.

The typical approach will be to run a small tournament between Nexplayer and other agents that represent the baseline that we compare against.

Unfortunately the results of such a tournament are influenced by a vast number of parameters. Most influential among these are

Game a high number of games should be played to demonstrate generality of the agent

Role Setup for each game, each agent-to-role assignment should be played an equal number of times to reduce role-specific effects

Time Control different time controls should be used to evaluate general playing strength as well as playing strength in a trade-off against time

Matches a number of matches should be played to obtain reliable results per game and role setup

Opponents an evaluation should be performed against different opponents

Assuming for each dimension just three discrete categories (i.e., three games, three roles, ...), we end up with $3^5 = 243$ matches for evaluating a single matter. Still critics could say that three games is far from general, that the time controls favor a specific opponent, that three matches is too few and that some opponent X should have been included because it would have radically changed the results. The fiercest among the critics would add that the search algorithm should be part of the test dimensions and that each individual optimization should be considered in a separate test setting.

By now it should become clear that a completely satisfactory evaluation is practically impossible.

We address this problem by adhering to the following guidelines within the evaluation of single topics.

Two-Player Games We will only play two-player games[3]. The reason is that the results of single-player games are difficult to compare and those of multiplayer games much less stable. Besides, for two-player the selection of search algorithms is less disputed and reduces the influence of search on the results. The list of games used can be found in the appendix A.1.2.

Preselected Games Single topics are evaluated using a set of games that match the topic. In this way, we avoid, e.g., evaluating distance features on games where distances are not relevant.

[3] with the exception of Pac-Man where the third player is defined as random player and has, due to the game mechanics, virtually no influence

Switching Roles As a general rule, we switch roles between the two agents after each match. Naturally, we play each game an even number of times so that each agent can occupy each role the same number of times.

Single Time Control A topic is evaluated using a time control that enables the application of the subject.

Fluxplayer as Opponent For evaluating a topic, we will only use Fluxplayer [ST07, Sch11]. This avoids a number of problems such as search parameters as Fluxplayer uses the same code base as our agent. Besides, Fluxplayer is also well known and one of the agents constantly ranging among the top agents of the annual competitions.

Gradual Improvement For each single approach proposed we enhance our agent by a module that implements the corresponding approach. Other evaluations in this work can thus automatically serve as a context for interpreting the results.

Full Optimization We apply all optimizations as soon as their preconditions allow it.

In short, the purpose for the evaluation of each topic is to demonstrate improvement. We make up for the lack of generality in the last chapter of this thesis where we drop most of the guidelines in favor a comprehensive full-scale evaluation.

2.6.1. Comparisons with the Win Rate

In order to demonstrate gradual improvement, we often need to compare the results of various setups of one agent against another. To keep these comparisons simple, we reduce the result of each setup to a single number which we refer to as "win rate".

The win rate of an agent is the percentage of points achieved by that agent relative the total of points achieved by any agent. This means that the win rate for an agent is 0 if the agent did not receive any point and it is 1 if it achieved all points. In a two-player game, a win rate of 0.5 indicates that both agents perform equally well in a setup.

Formally, given some setup and a set of agents A where each agent $a \in A$ achieved a total of $t(a)$ points, the win rate wr is

$$wr(a) \stackrel{\text{def}}{=} \frac{t(a)}{\sum\limits_{a \in A} t(a)}$$

Naturally, if a setup includes more than one game or match, we first aggregate the individual agent points and only calculate the win rate as last step.

Since most of games we evaluate on are zero-sum games, the win rate is equal to the average goal value achieved divided by 100. If no agent achieved any point, the win rate is undefined. We will explicitly address this case, should it occur.

2.6.2. Comparisons with States per Second

In order to demonstrate the effect of different approaches on run time, we often compare the states evaluated per second. The value is the number of different states evaluated during a match divided by the duration of the match. If a state is encountered more than once, its value will be obtained from a cache, avoiding (the cost of) evaluation thereby. The numbers were obtained on a 2.9GHz AMD A8-3850 with 4 CPUs and 8GB RAM.

For a complete specification of the test environment, see the appendix A.1.

3. Evaluation Functions I - Aggregation

In this chapter we aim to determine how an evaluation function should look like. To separate concerns, we will focus on the general structure of evaluation functions, that is, types of aggregation functions and how these can be used to aggregate the output of features. We will treat features themselves as a separate subject and dedicate chapter 4 to feature detection, construction and integration.

Building on the criteria of evaluation functions (section 2.5), we start by analyzing different approaches for building evaluation functions with focus on aggregation functions.

We proceed in section 3.2 by presenting an algorithm for constructing an evaluation function based on neural networks. The algorithm allows for the initialization of neural networks using logic programs and qualifies as optimal with respect to our desired function properties.

In section 3.3 we address a major problem of the neural network approach, enabling us to fully benefit from its potential.

3.1. Choice of Evaluation Function

We start by briefly analyzing currently applied probabilistic and deterministic evaluation functions (see section 2.3) and discuss how they relate to the criteria established in section 2.5.

3.1.1. Reasons against Probabilistic Evaluation

The construction of a probabilistic evaluation function is efficient as its requirements are similar to the requirements of playing a general game. These are specifically the determination of

- legal moves
- state updates
- terminal
- goal

Consequently, the evaluation function is constructed "instantly" and its representation is bounded by the size of the game description. For evaluating the function, we must determine the terminality of a state, a legal move and a state update for each state we

encounter. Thus the computational cost depends on the depth of the remaining game tree.

With respect to quality, single runs of a probabilistic evaluation functions are unlikely to be consistent with each other. Therefore, probabilistic functions are usually employed together with UCT search that performs repeated evaluations. The repeated evaluations gradually obtain consistency, however, to the detriment of efficiency.

The state value approximated is not the expected value given optimal play as assumed by a perfect evaluation function $v^*(s)$ but the expected value given random play which we refer to with $pv(s)$. Therefore strict monotonicity does not hold generally.

Still, there are many cases where it may be assumed to hold due to $pv(s)$ being approximately equal to $v^*(s)$, that is, $pv(s) \approx v^*(s)$:

One reason is that for advanced (deeper) states in the game the terminal state is closer and therefore the noisy effect of random moves gradually disappears. This means that state values vary less and evaluations are faster, thereby enabling more and more accurate evaluations in a constant amount of time.

The second reason is that in practice, there are many cases in which a sufficient degree of uncertainty holds for $pv(s) \approx v^*(s)$ to hold. The uncertainty is rooted either in opponent behavior or ambiguous value backpropagation in the search algorithm. As a consequence, players cannot make informed decisions and therefore do not significantly deviate from the random-move assumption. Frequent instances of such situations are

- matches with uninformed players (opponent uncertainty)
- games with many players (opponent uncertainty)
- simultaneous move games (opponent + value uncertainty)
- games with many Nash equilibria per state (value uncertainty)

In summary, the compactness, ad-hoc construction and extrapolation properties generally hold. All other properties depend on the game and cannot be considered to hold generally: While strict monotonicity may hold, the major problem of probabilistic evaluation functions is their inefficiency. This inefficiency is a direct result of its run time depending on the size of the remaining game tree and the need for repeated evaluation to obtain consistency.

While extensions may mitigate some of these disadvantages (see section 6.1.1 for an overview), the probabilism itself and the requirement for repeated evaluations is not eliminated.

3.1.2. Reasons for Deterministic Evaluation

Deterministic evaluation functions are usually designed as functions operating on subfunctions (features) that in turn operate on the state. As such, they are compact to represent and the upper bound on computation costs is determined by the maximum upper bound of each of its features. Therefore, typically the most expensive features are discarded unless they prove of significant importance for state evaluation.

Deterministic evaluation functions are trivially consistent.

Linear Combinations of Features

Linear combinations of features are sums of weighted features and can be used directly as evaluation function (e.g., in Kuhlplayer [KDS06] or Ogre [Kai07c]) or constitute a part of it (as in Cluneplayer [Clu07]).

Regardless of how features are determined, they must be assigned weights. If the same weights are assigned for each feature (as done in Ogre) or the weights are assigned according to a ranking (as in Kuhlplayer), relations between the importance of features are ignored and we can expect a weak quality of the evaluation function in general.

On the other hand, if better weights are to be determined, empiric data obtained by simulating random matches is used (Cluneplayer). The process itself is time-consuming, and we can expect the evaluation function to improve for higher start clocks due to finding more features or weighting them better. Then, however, the ad-hoc construction criterion is in a trade-off with the approximation quality of the agent's evaluation function. However, even given infinite time, the quality criteria cannot be guaranteed. In fact, depending on the features and weights determined, it is possible that the $v^*(s)$ cannot be sufficiently well approximated by the evaluation function. This is typically the case in non-trivial games.

Fluxplayer

Fluxplayer [ST07, Sch11] is the only agent described in literature that does not rely on a linear combination of features but transforms the goal rules to an evaluation function instead. The construction time for the aggregation function is thus approximately bounded by the size of the goal description. Feature detection is restricted to expressions occurring in the goal condition and the features can substitute the expression they are derived from, eliminating the need for weighting features. Consequently, the evaluation function of Fluxplayer fulfills the ad-hoc construction criterion.

From a qualitative viewpoint, the evaluation function is a correct fuzzified version of the goal function and thus fulfills the extrapolation criterion. Since the fuzzy aggregation function itself is strictly monotonic and correct, it also imposes a strict order on non-terminal states that aligns with the order imposed by the perfect evaluation function *if* state representation and goal rules reflect this order. Under the same condition, symmetric states are identified and evaluated to equal values.

We conclude that all qualitative properties hold *if* the goal condition reflects the valuation. Fluxplayer is the only agent that fulfills all efficiency criteria and can be expected to fulfill the monotonicity criteria in some games.

3.1.3. Shortcomings of Deterministic Evaluation Functions

We discussed the properties of currently used evaluation functions and determined their strengths and weaknesses. Most criteria can be easily assessed with the exception of strict monotonicity which can be only evaluated given the perfect evaluation function that in turn depends on the game.

Since there it is hard to determine whether an evaluation function is perfect (or optimal), we must assume that our evaluation function is imperfect.

Consequently, we should apply measures to improve the evaluation function. We propose a continuous evolutionary process for this purpose. The rationale is that a repeated adjustment of the function parameters towards high-quality evaluations eventually lets the function converge towards (a better approximation of) the perfect evaluation function. This can be achieved using learning.

In order successfully apply learning techniques, our function must exhibit additional properties:

Generality The structure of our evaluation function must be general in order to capture the perfect evaluation function.

Adjustable Parameters The function must possess a sufficient number of free parameters to be able to capture the perfect evaluation function.

Initialization A good initialization increases chances for convergence and speed of convergence of our function to the perfect evaluation function. The ad-hoc construction criterion requires the initialization to be quick.

All three requirements pose separate problems.

Generality is a structural question that requires a function to be flexible enough to represent the perfect evaluation function. This trivially rules out all evaluation functions that rely on a specific structure such as Cluneplayer's lottery model. Also, linear combinations of features are only able to represent non-linear functions if their features are appropriately designed. Given a lack of general and good acquisition algorithms of important features, we may safely assume that linear combinations of features are generally not able to represent arbitrary functions. The subject of feature acquisition is discussed in detail in chapter 4.

Adjustable parameters is a closely related problem that requires that the parameters in the evaluation function be useful and sufficient in number to actually model the perfect evaluation function. The problem is different from generality as a general function can have most of its parameters fixed. Specifically, this is a problem of Fluxplayer since, for example, there is no native way to change feature weights.

Finally, we need an initialization of all parameters that is as close as possible to the perfect evaluation function. Randomly initialized weights do not adhere to quality principles while learned weights require too much time to be used in a GGP agent.

None of the current agent systems employs a function expressive enough to approximate arbitrary functions. Most use more or less adjustable parameters with the exception of Fluxplayer that in turn is the only agent with a convincing strategy for initialization.

3.1.4. Conclusions

Obviously, no single aggregation function fulfills the criteria established throughout this section and in section 2.5. Each single approach has its strengths and weaknesses, but all approaches fail when trying to gradually approximate the perfect evaluation function.

An aggregation function that will be able to fulfill all the criteria and consequently perform better than the current agent systems will try to unite the strengths of each single approach. This entails decisions in favor of

- a deterministic evaluation function in order to obtain control over state evaluation
- an initialization based on the goal function comparable to that used in Fluxplayer (in contrast to a semi-learning approach)
- an evaluation function capable of learning

3.2. An Aggregation Function Based on Neural Networks

The aforementioned properties can be realized using neural networks. Neural networks are deterministic functions that can be easily trained, and a subset of these networks are able to approximate arbitrary functions to an arbitrary degree. Therefore they are a good representation for an evaluation function.

A point against them is the initialization problem of neural networks: A good initialization defines the network with respect to its structure, quantity of neurons and their connection weights. However, given the vast number of parameters, finding good initialization parameters is difficult [KPKP90, TF97], resulting thereby in bad approximation, the need for lengthy training or non-convergence.

To address this problem we will use the $C - IL^2P$ algorithm (*Connectionist Inductive Learning and Logic Programming System*, [dG02]). Originally developed for representing propositional logic programs as neural networks, we modify $C - IL^2P$ to represent a fuzzy evaluation of such a program. By initializing the neural network with the propositionalized goal rules, we obtain thus a fuzzy evaluation of the goal condition. Still, we retain all the benefits of a neural network while solving at the same time the initialization problem.

The remainder of this section is structured as follows: In the next section we will briefly review neural networks and $C - IL^2P$ that are the tools for our algorithm.

In section 3.2.2 we proceed with a step-by-step description of the transformations needed, starting from a goal description and ending in a state evaluation function.

In section 3.2.3 we discuss how to handle the practical limitations of our approach and evaluate it in section 3.2.4. We conclude with a small summary in section 3.2.5.

The remainder of this section is a revised version of our paper "Neural Networks for State Evaluation in General Game Playing" [MT09], presented at ECML 2009.

3.2.1. Preliminaries

Our aim is to initialize a neural network by applying the $C - IL^2P$ algorithm on the propositionalized goal condition of game.

For this purpose, we will briefly review neural networks, propositional domain theories and the $C - IL^2P$ algorithm.

Neural Networks

Neural networks are a classic tool of AI research. They are inspired by the model of nervous processing in animals and are originally motivated by the belief that a reconstruction of a brain exhibits the same properties as a brain, i.e., intelligence. Due to the complex mechanism of how brains work, research was never able to reproduce "intelligence" using a neural network. The overly ambitious goals were thus abandoned in favor of a simple and practical model of a function approximator based on a simplistic model of the brain.

A neural network is a groups of interconnected atomic units, called neurons. There are numerous types of neural networks and for a complete treatment of the topic we recommend [Bis95]. However, in this work we will only use plain old multilayer feed-forward networks and thus focus on these exclusively. Here, neurons are organized into several layers where each neuron in any layer has an outgoing connection to each neuron in the following layer. The network is thus a directed acyclic graph. The first layer of neurons is called the input layer and the corresponding neurons input neurons. The neurons in the last layer are called output neurons.

Any neuron y is an n-ary function with a single output o_y based on the output of its predecessor neurons o_{x_i}.

$$o_y = h\left(\sum_{i=1}^{n} w_{x_i y} o_{x_i}\right)$$

The $w_{x_i y}$ are the connection weights, and the function $h(x)$ is the activation function.

Typically, the activation function maps to the unit interval (unipolar network), or the interval $[-1, 1]$ (bipolar network).

Network training is performed using error backpropagation and in this process, the weights w_{xy} from the neuron x to the neuron y are modified.

Given that three-layer feed-forward neural networks are universal function approximators, they are often the algorithm of choice for approximating and modeling an unknown function.

Propositional Domain Theory

In the remainder of this chapter we will use propositional domain theories as a simplified form of knowledge representation without first-order variables. While they are semantically equivalent to propositional logic programs, their tree-like representation is isomorphic to the networks we produce, making them an intuitive intermediate step.

We consider a propositional domain theory a set of rules of the form

$$q \Leftarrow \bigotimes_{1 \leq i \leq k} p_i \text{ with } \bigotimes \in \{\bigwedge, \bigvee\}$$

where q is an atom and all p_i are literals (an atom or a negated atom). Depending on the operator $\bigotimes$, we interpret each rule as a conjunction or disjunction of its k antecedents. Without loss of generality we assume that there are no two rules in the domain theory with the same atom occurring in the head.

Consequently, each propositional domain theory (short: domain theory) can be represented as a directed graph where there is a single node for each atom in the domain theory and there is a connection between two atoms p and q if there is a rule where q occurs in the head and p occurs in the body. The connection is labeled negative if p occurs as negative literal in the body of the rule, else it is labeled as positive. Each node can be labeled a conjunction or disjunction if there is a rule with the corresponding atom in the head. If there is no such rule, we label the node as input node.

The resulting graph depicts for each node q the subproofs p_i needed to proof q. By recursively proving each p_i we can construct the complete proof in terms of the atoms of input nodes.

Relation to Propositional Logic Programs Typically, information encoded in propositional logic is represented as a propositional logic program, that is, a set of clauses of the form

$$A \Leftarrow L_1, .., L_n$$

where A is an atom and L_i with $1 \leq i \leq n$ are literals (an atom or a negated atom). A is then interpreted as a conjunction of all L_i.

The main difference is the representation of disjunctions: While in a domain theory we represent a disjunction explicitly as rule, propositional logic programs encode them implicitly by interpreting each atom A as true if there is a true clause with A as head.

We choose the domain theory interpretation for two reasons: First, by referring to each rule as disjunction or conjunction explicitly, we can simplify the further discussion of domain theories due to symmetries of the rules, that is, we do not need to pay attention to the structure of the representation. And second, we can assume, without loss of generality, that all atoms occur at most once in the head of the rule. This will allow us to refer to each atom unambiguously as conjunction, disjunction or input neuron and allow for symmetries between the domain theory graph and the neural network representing that graph.

Still, a domain theory is a slightly more general representation of a propositional logic program where disjunctions can be encoded in terms of negative literals. For our purposes, we treat both as equivalent.

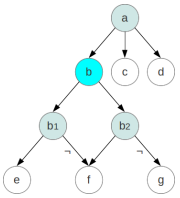
Logic Program	Domain Theory	Domain Theory Graph
$a \Leftarrow b, c, d$	$a \Leftarrow b \wedge c \wedge d$	
$b \Leftarrow e, \neg f$	$b \Leftarrow b_1 \vee b_2$	
$b \Leftarrow f, \neg g$	$b_1 \Leftarrow e \wedge \neg f$	
	$b_2 \Leftarrow f \wedge \neg g$	

Figure 3.1.: Three Equivalent Representations of a Propositional Logic Program. Nodes in the graph are colored depending on whether they are conjunctions, disjunctions or input neurons.

Figure 3.1 shows three equivalent representations of a propositional logic program. The graph uses three different colors to distinguish conjunction, disjunction and input nodes from each other.

In the remainder of this work we will exclusively obtain domain theories by deriving them from the goal rules of a game. Therefore, unless explicitly stated otherwise, the domain theories under discussion are finite, acyclic and connected and the corresponding graph possesses a unique root node. The graph is thus a rooted DAG, or, more conveniently, a tree.

The $C - IL^2P$ algorithm

$C - IL^2P$ (*Connectionist Inductive Learning and Logic Programming System*, [dG02]) is an algorithm that takes as argument a propositional domain theory and returns a neural network. Each atom in the domain theory is represented by exactly one neuron. A neuron z with the output o_z represents a propositional variable q via the following constraints:

$$q \Leftrightarrow o_z \in (A_{min}, 1] \qquad\qquad \neg q \Leftrightarrow o_z \in [-1, A_{max})$$

So if the atom q is true, the corresponding neuron gives a value of $o_z \in (A_{min}, 1]$ while an output $o_z \in [-1, A_{max})$ is interpreted as falsity. A_{min} is thus the minimum output of a neuron to be considered true while A_{max} is the maximum neuron output to be considered false. Outputs in the non-empty interval (A_{max}, A_{min}) represent the "zone in between" and are guaranteed to not occur. Usually, $-A_{max} = A_{min}$ is set, allowing propositional negation to be encoded as arithmetic negation. Since we will make no use of the recurrence represented by $C - IL^2P$, we will discuss a slightly simpler algorithm that avoids recurrence.

$C - IL^2P$ uses as input a propositional domain theory which is a set of rules of the form

$$q \Leftarrow \bigotimes_{1 \leq i \leq k} p_i \text{ with } \bigotimes \in \{\bigwedge, \bigvee\}$$

where q is an atom and the p_i are positive or negative literals.

Each rule is represented by one neuron for the head and k neurons for the body. The weight between the body neurons and the head neuron is W for a positive antecedent p_i and $-W$ for a negative antecedent. Additionally, the neuron for q is subject to a threshold, represented by a connection to the bias unit (with constant output 1) with the weight $\theta_\wedge$ ($\theta_\vee$) if the rule is a conjunction (disjunction) as given in equation (3.2.1).

The parameters W and θ are defined as follows:

$$\theta_\vee(k) = -\theta_\wedge(k) = \frac{(1 + A_{min}) * (k - 1)}{2} * W \tag{3.2.1}$$

Both W and A_{min} can be set freely but are subject to the following restrictions:

$$\frac{k_{max} - 1}{k_{max} + 1} < A_{min} < 1 \tag{3.2.2}$$

$$W \geq 2 * \frac{ln(1 + A_{min}) - ln(1 - A_{min})}{k_{max}(A_{min} - 1) + A_{min} + 1} \tag{3.2.3}$$

k_{max} is the maximum number of antecedents a rule has.

The result is a neural network with output values in $[-1, 1]$ where all input neurons represent propositional variables and all other neurons represent either a conjunction or a disjunction.

Alternatives to $C - IL^2P$

There are various alternatives for representing logic in the form of neural networks.

KBANN (Knowledge-Based Artificial Neural Network, [TSN90, TS94]) used unipolar neural networks (with output in $[0, 1]$) for representing truth. Its main problem was a restriction on the number of antecedents.

A comparable approach with focus on representing propositional logic programs was proposed in [HK94]. It was shown that three-layer recurrent neural networks can represent propositional logic programs without any restrictions regarding the number of rules or antecedents. Learning, however, was not supported due to the use of discrete threshold units.

The problems of both were addressed by $C - IL^2P$ [dCLd96, dG02].

Recent approaches suggest to represent first-order logic directly in a network, avoiding propositional logic altogether (see [BH05, Bad09] for an overview and discussion). However, we do not use any of these approaches for several reasons.

First and foremost, a state is represented in GDL as a ground and finite set of finitely nested terms. States are thus essentially propositional and would not benefit from a first-order representation of the network. In contrast, networks using first-order logic represent atoms in an interpretation as vectors, creating considerable overhead for state evaluation. Finally, for purely practical reasons a well-established and simple approach such as $C - IL^2P$ is sufficient for our purposes.

3.2.2. From the Game Description to an Evaluation Function

Given a role r with the associated goal values $\{gv_1, \ldots, gv_n\}$ we obtain a set of networks with the output neurons $\{o_{r,gv_1}, \ldots, o_{r,gv_n}\}$ by iterating over all goal values. For each goal value gv we

1. transform the goal rules for `goal(r, gv)` to a propositional domain theory via expansion and ground-instantiation
2. transform the propositional domain theory to a neural network with the output neuron $o_{r,gv}$ using $C - IL^2P$

Finally, we define the state evaluation of a state s for role r as the aggregated output over all networks.

We proceed by explaining each step in detail.

Goal Clauses to a Propositional Domain Theory

Consider the set of goal clauses of the form `goal(r, gv) :-conditions` for a given role r and goal value gv as defined in the rules of some game.

We expand and ground-instantiate the set of goal clauses with the following algorithm:

1. Let ϕ be the set containing the expression `:- goal(r, gv)`
2. Repeatedly add all rules whose head unifies with any literal in the body of any rule in ϕ until ϕ stabilizes.
3. While ϕ contains a rule r with a variable v, remove r from ϕ and add all instances of r where v is substituted by its possible ground-instances.

For the sake of simplicity, we assume that expansion and ground-instantiation are not limited by our hardware. However, this is oversimplified and we will treat technical problems that may arise in section 3.2.3.

The result of the algorithm is a set of ground rules without variables, i.e., a propositional logic program. We transform the program to a domain theory by explicitly introducing a disjunction for each atom that occurs in the body at least one rule and in the head of at least two rules. An example is given in figure 3.1.

The resulting domain theory can be described as a tree where each node is an expression. The root of this tree is the expression `goal(r, gv)`, all non-leaf nodes are either a conjunction or disjunction of its children, and all leaf nodes query the state with the GDL keyword `true/1` or represent state-independent facts. Negation is treated as a label attached to the edge between two nodes.

Since we are interested in evaluating the state as fast as possible, we can simplify the propositional domain theory further by evaluating state-independent facts (such as `successor(1,2)` or `distinct(a,b)`) immediately to true or false. Naturally, other nodes of the domain theory may become generally true or false as a consequence and the domain theory can be further simplified. We assume in the following a simplified domain theory where only state-dependent expressions are leaf nodes of the domain tree graph.

A Domain Theory to Neural Networks

We apply $C - IL^2P$ to the domain theory to obtain a network that is isomorphic to the domain theory graph. For this purpose we apply the following algorithm:

1. Obtain k_{max} as the maximum number of antecedents that a rule has.
2. Determine A_{min} as given in equation (3.2.2).
3. Determine W as given in equation (3.2.3).
4. Create a network and for each atom a in the propositional domain theory add a neuron n_a to the network.

5. Add a neuron *bias* to the network.

6. For each rule r in the domain theory do
   - Let h be the atom in the head of the rule
   - For each literal b in the body of the rule do
     - If b is a negative literal, add a connection from n_b to n_h with weight $-W$
     - If b is a positive literal, add a connection from n_b to n_h with weight W
   - Depending on whether r is a conjunction/disjunction and the number of antecedents k of rule r, add a connection from *bias* to n_h with weight θ according to equation (3.2.1).

Since A_{min} and W are only given as inequalities, we recommend to set the weight W to its possible minimum. For a discussion of the reason we refer the impatient reader to section 3.3. We empirically determined the weight W to be minimal if we set A_{min} to $0.8A^*_{min} + 0.2$ where A^*_{min} is the minimum as given by equation (3.2.2).

The application of the above algorithm produces a network that is similar to the underlying domain theory: For each node in the graph there is exactly one neuron in the network and a neuron is connected to another if the corresponding atoms occur in the body and head of the same rule. Each connection between two neurons has a weight attached as defined by $C - IL^2P$ and logical negation is encoded by arithmetic negation of this weight. If a neuron has no incoming connections, then it is an input neuron.

The only structural difference between the domain theory graph and the neural network is that, in addition to the connections from a node to its children, each non-input neuron has a connection to a bias neuron.

For evaluation of a state, we set all input neurons to output 1 if the atom represented by the neuron holds in the state, and to -1 otherwise. The output value of all other neurons can be calculated based on these. Each neuron responds with a real number in the interval $[-1, 1]$. Specifically, the output value o of a neuron is in the interval $o \in (A_{min}, 1]$ if the corresponding atom holds and $o \in [-1, -A_{min})$ otherwise.

This implies that the output neuron of the network representing the root `goal(r, gv)` of the tree returns a positive value if the state satisfies the domain theory and a negative value otherwise. More importantly, due to the monotonicity of the network, we can assume that the better the domain theory describes the current state, the higher the network output is.

Aggregating Neural Network Outputs

For any role `r` and goal value `gv` we obtain a network whose output $o_{r,gv}(s)$ for a state s indicates the extent to which the state s fulfills the corresponding goal condition. We aggregate these values for role `r` to obtain a single state value $v_r(s)$.

$$v_r(s) = \gamma + \sum_{gv} (gv - \gamma) * \frac{o_{r,gv}(s) + 1}{2}$$

To relate the network outputs to our state value, we use a standard goal value γ that represents the standard goal value we expect to achieve. We interpret each network output as a degree to which we should deviate from the standard goal value. Since each role has exactly one valid goal value, we can assume that we do not fall out of the goal value interval $[0, 100]$. However, to ensure this, we map all values v less than 5 to $5 + \frac{gv - 5}{100}$ and similarly all values greater than 95 to $95 + \frac{gv - 95}{100}$. Thus we sacrifice resolution in these cases in order to ensure that the $v_r(s)$ is always in $[0, 100]$. Though this may seem a restriction, the situation occurs very rarely and does not affect strict monotonicity.

An advantage of the above approach is that for $gv = \gamma$ the difference in the above formula becomes zero and we do not need to evaluate the corresponding network. In addition, by setting γ we may bias our evaluation optimistically by setting a high γ. The evaluation function then returns high values unless one of the networks returns a high certainty that another goal value should hold.

3.2.3. Issues and Optimizations

While the algorithm will work in most of the cases, there are problems that need to be dealt with.

Most importantly, during expansions and ground-instantiation of the goal clauses the size (i.e., the number of rules) of the domain theory may rise exponentially in the number of variables. This is undesirable as it potentially increases the time for both construction and evaluation of the function exponentially. There are several subcases which we treat differently:

Patterns with Exponentially Many Instantiations Some patterns in the definition of rules can be found that are known to produce an exponential number instantiations. We avoid these patterns altogether by substituting the corresponding expression by a ground dummy predicate and avoiding its ground-instantiation thereby. We evaluate the dummy predicate by delegating the evaluation of the original expression to the reasoner. Construction is thus skipped, the network size constrained and evaluation is still fast since we evaluate the expression without the overhead of a neural network. Specific examples of such patterns are recursive predicates and conjunctions with variables shared across conjuncts such as $p(0, A) \wedge p(A, B) \wedge \ldots \wedge p(Y, Z)$.

General Case Although the above patterns represent a good heuristic to avoid combinatorial explosion, they are not sufficient. Therefore we impose artificial size constraints. Once violated, all expressions with variables are substituted by ground dummy predicates and delegated to the reasoner. We currently allow a maximum of 512 neurons for all networks.

We use other strategies that do not only limit the scope of the above problem but also optimize the algorithm:

Reuse of Neurons Since atoms occur often in several rules, we reuse their corresponding neurons. We also reuse neurons across networks, that is, if multiple goals depend on the same atom, then their networks use the same neuron.

Lazy Expansion / Instantiation We create the neural network in breadth-first order, starting from the goal clauses, and only expand and instantiate expressions level by level. In this way we ensure that size constraints are only violated in less important subtrees far away from the root. Also, the costs for ground-instantiation are only incurred when necessary.

Heuristics for Instantiation We prefer instantiation of state-independent predicates to instantiating variables randomly. The reason is that variable instances are often heavily constrained by these predicates.

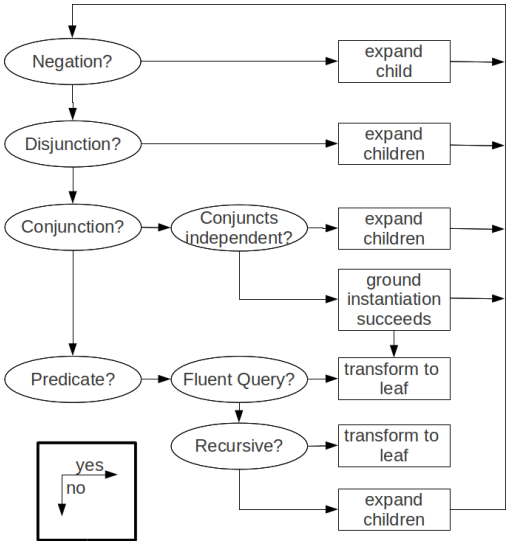


Figure 3.2.: Data Flow of Lazy GDL Formula Expansion

Figure 3.2 shows the data flow of expanding a GDL formula lazily. The graph is a decision graph where conditions are shown as ellipses with a question mark and actions are shown as rectangles. If a question is answered with "yes" the data flow is indicated by an arrow to the right, otherwise by an arrow to the bottom.

Neurons are created in three cases:

- A conjunction neuron is created for each formula that can be represented as a conjunction of other formulas.
- A disjunction neuron is created for each formula that can be represented as a disjunction of other formulas.
- A formula is transformed to a leaf node that will later serve as input neuron.

There are some simplifications in the graph that deserve special attention. The condition tagged "conjuncts independent" checks whether the conjuncts of a conjunction do not share variables. If they share variables, they are not independent and we try to ground-instantiate them with a timeout. If that is not possible, we transform he expression to a leaf. The condition "fluent query" checks whether the predicate is of the form $true(p)$ and transforms p to an input neuron whose value depends on whether p holds in a given state or not.

3.2.4. Evaluation

The evaluation function based on neural networks created with $C - IL^2P$ can be considered generally advantageous. It is compact and computation time is approximately linear in the time for evaluating the goal condition. Trivially, the function is consistent.

More importantly, however, the evaluation function inherits the strict monotonicity of the underlying networks that in turn inherit it from the strict monotonicity of each neuron. Consequently, changes in the input produce changes in the output. Thus, if the structure of the goal condition reflects a useful evaluation, we can assume that the strict monotonicity criterion is fulfilled in these cases. Likewise, strict monotonicity with respect to symmetries can be assumed if the domain theory exhibits symmetry (via commutative conjunctions and disjunctions).

Also, the evaluation function can be produced directly from the ground-instantiated goal rules, making their creation ad-hoc and their evaluation a fuzzy extrapolation of the goal theory.

Consequently, with the exception of general strict monotonicity, all our basic criteria are fulfilled.

Given that the function is itself general and has adjustable parameters, neural networks may prove a competitive tool for state evaluation.

We will evaluate this assumption in experiments.

Experiments

Our hypothesis is that given our neural evaluation function, our agent should be able to compete with Fluxplayer in games that are tailored for a fuzzy evaluation, that is, where a plain fuzzy evaluation of the goal rules provides a good heuristics.

Since most other games can be expected to provide at least a partially descriptive goal condition, we can assume that our evaluation function generally contributes positively to the agent's playing strength. In turn, should our agent not be able to compete with Fluxplayer, the approach should be revised.

We test our hypothesis by setting up our agent Nexplayer against Fluxplayer in the subset of games where we believe a fuzzy evaluation of the goal condition is presumably a good heuristics. These games are typically variations of Tic-Tac-Toe and Connect Four since their goal conditions are essentially disjunctions of conjunctions of fluents. For each game, we play 20 matches with 40 seconds start clock and 4 seconds play clock.

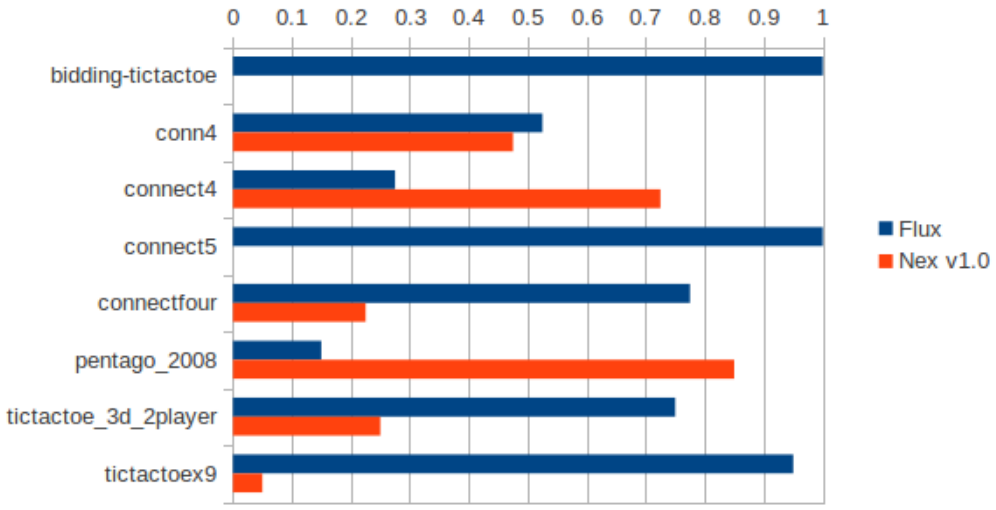


Figure 3.3.: Win Rate of Nexplayer v1.0 against Fluxplayer

Figure 3.3 shows the average points achieved per match. Nexplayer wins in average 32.2 points while Fluxplayer wins 67.8 points. The statistics give some insight into the benefits of the evaluation function.

Beginning with the worst, it seems that in three games (`bidding-tictactoe`, `connect5`, `tictactoex9`) Nexplayer does not stand any chance against Fluxplayer. In all other games Nexplayer achieves a few victories, indicating that in some situations the heuristic is acceptable. We interpret this as a sign that while the quality of the heuristics of Fluxplayer is superior, it is not out of reach for Nexplayer.

Obviously, `connect4` and `pentago_2008` are the only games where Nexplayer outperforms Fluxplayer. This may have several reasons. Fluxplayer may not have been able to initialize all features given the tight start clock or its heuristics is simply bad.

Figure 3.4 shows the average number of unique states inspected per match. As we can see, Nexplayer is constantly faster in its evaluation and evaluates in average three times the number of states of Fluxplayer. Assuming that the majority of states repeats itself, this should roughly translate to Nexplayer searching a little less than an additional level of states. This may have several reasons, most importantly the fact that Nexplayer did not spend time on creating or evaluating features.

3.2.5. Conclusions

We have presented an algorithm that allows to automatically derive an evaluation function based on the goal condition. The function fuzzy-evaluates states and can be trained using standard learning algorithms such as backpropagation. The exact details of the training procedure are discussed in our publication [MT09] and were omitted here due

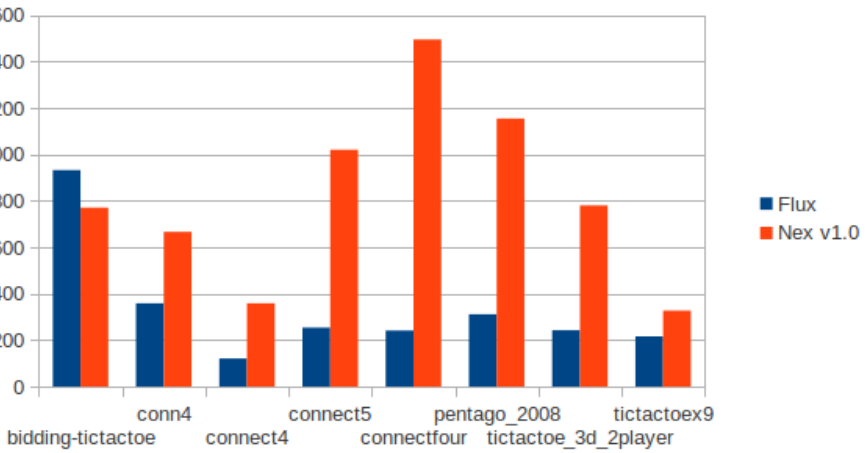


Figure 3.4.: Nex v1.0 vs. Flux - States per Second

to a focus on the evaluation function construction. Nevertheless, our evaluation function adheres to the criteria established in our prior analysis in the sections 3.1.4 and 3.1.3.

While all practical problems could be resolved, the evaluation showed an underperformance of Nexplayer compared to Fluxplayer. Yet results were promising as Nexplayer won a few matches, indicating that its heuristics is not far off in terms of quality.

After analysis of the results, we found that one of the reasons for the weak performance of Nexplayer was a problem which we call the resolution problem. The problem is related to $C - IL^2P$ and the fact that is was not designed for creating fuzzy evaluators from domain theories. More specifically, the $C-IL^2P$ translation of a rule with k antecedents leads to an absolute activation a of up to approximately $a = |\theta + k * w| \leq |2kW|$. Thus for high k activation becomes high and the absolute output $|o| = |h(a)| \approx 1$. While this itself is not a problem, limited resolution of the standard floating-point representation may make two states seem similar though they are different.

Translated to GGP, different input states may lead to equal outputs of the evaluation function, violating thus the strict monotonicity property. This happens especially if the neuron has a high number of predecessor neurons and the absolute activation for the two different states is high. We will address this problem in the next section by specializing $C-IL^2P$ such that it preserves resolution to a much higher degree and is thus generally better suited for creating fuzzy-evaluation networks.

3.3. High-Resolution State Evaluation using Neural Networks

The algorithm presented in section 3.2 transforms the goal function of a game into a set of neural networks using $C-IL^2P$. These networks can then be used for fuzzy evaluation of states. While the algorithm is theoretically well-suited for fuzzy evaluation, a practical limitation may compromise its utility as the following example demonstrates:

Example 3.1. Consider the game `connectfour`, a Connect Four variant played on an 8x6 board. After initializing the agent, a 3-layer network with 104 leaf nodes is created. Our agent is first to move and examines the eight possible moves `drop(1) .. drop(8)` indicating the column where the disk is to be dropped. For each move, we generate the successor state that contains a single fluent `cell(X, 1, red)` relevant for evaluation where `X` indicates the column where the disk was dropped. After evaluating all 8 states, we obtain the following results:

	successor state evaluation	
move	role `red`	role `black`
`drop(1)` or `drop(8)`	32.9684512447111	32.9684512447111
`drop(2)` or `drop(7)`	32.9684512447111	32.9684512447111
`drop(3)` or `drop(6)`	32.9684512447111	32.9684512447111
`drop(4)` or `drop(5)`	32.9684512447111	32.9684512447111

Table 3.1.: $C - IL^2P$ Evaluation of all ply 1 States of `connectfour`

The values are not artificially rounded. So each state is evaluated to the same value, although theory states that `drop(4)` and `drop(5)` should be the better moves because they occur in more possible lines of four. What happened?

To track down the problem, we construct a smaller problem with the same characteristics.

Example 3.2. Consider a neuron z representing a conjunction of the output of 4 preceding neurons $z \Leftarrow \bigwedge_{i=1}^{4} y_i$. We assume the maximum number of children of a node to be $k = 4$ and set $A_{min} = -A_{max} = 0.9$ and $W = 4$, exceeding thus their theoretical minima 0.6 and 3.93 as required by equations (3.2.2) and (3.2.3). The following table shows the neural output of z for t of the 4 y_i representing *true* (having output 1). While

true antecedents t	activation a_z	output o_z
0	-27.4	-1.000000
1	-19.4	-1.000000
2	-11.4	-0.999978
3	-3.4	-0.935409
4	4.6	0.980096

Table 3.2.: Neural Output of a Conjunction with 4 Antecedents

the neuron correctly encodes a conjunction, it is not able to distinguish the cases where

none or one antecedent is fulfilled. This is due to the resolution imposed (10^{-6}), but occurs similarly in all finite floating-point representations such as those used in 32-bit computers.

The latter example demonstrates that the given resolution of 10^{-6} is not sufficient to describe the differences of the output of neuron z for none or one fulfilled antecedent in a simple 2-layer network. The reason for this is the high absolute activation a_z (-19.4 and -27.4) that becomes approximately -1 after applying the activation function.

The lack of distinction in example 3.2 causes the lack of distinction in example 3.1. While there still exist minor differences at the first output neuron, in `connectfour` a three-layer network is used. Small differences in the output in the second layer get squashed further in each additional neuron layer due to the activation function. Eventually, the difference becomes negligible, rendering all ply 1 states for the `connectfour` network indistinguishable.

There are several ways to deal with the problem. Following the path of least resistance, we could use high-precision arithmetics. Unfortunately, high precision arithmetics push the solution of the problem to run-time. Since we use the evaluation functions to evaluate hundreds or more states per second, the additional overhead is unacceptable.

The second and much more viable alternative is to look in the $C - IL^2P$ algorithm for candidates that unnecessarily increase the absolute activation a_z.

$$\begin{aligned} a_z &= \text{bias} + \text{max. input} \\ &\leq \left| \frac{1 + A_{min}}{2} * (k-1) * W + k * W \right| \\ &< (2k-1) * W \end{aligned}$$

With the absolute activation bounded by $a_z < (2k-1) * W$, the only two parameters eligible are the number of children k and the weight W. While k cannot be influenced directly without changing the domain theory, $C - IL^2P$ can be modified to allow for smaller weights W. In particular, the global setting of W and A_{min} depends on the maximum number of antecedents of a rule k_{max} and implies thus weights higher than necessary for nodes with $k < k_{max}$ children.

In the remainder of this section we will present a modification of $C - IL^2P$ that exhibits lower weight requirements while preserving correctness. It is a revised version of our work "Neural Networks for High-Resolution State Evaluation in General Game Playing" [Mic11b], presented at the GIGA workshop 2011.

3.3.1. Local Neuron Transformation

We present a more general version of $C - IL^2P$. To maintain correctness, two basic conditions must be fulfilled when representing propositional logic as neural network. The first is the ability to correctly represent boolean operators, in this case the operators *and*, *or* and *not*. And the second is the distinction of truth and falsity at all levels during information processing. Both concepts, however, are related to different parameters of a

neuron enabling us to define them independently and allowing thereby the development of an algorithm that sets parameters locally (i.e., for each neuron).

We begin by defining a *standard neuron*.

Definition 3.1 (Standard Neuron)**.** Let z be an input layer neuron with no predecessors and the output value $o_z \in O \subset [-1, 1]$ *OR* a neuron with

- a real weight w_z,
- a real bias $bias_z$,
- and the unbiased input
$$\hat{\imath}_z = \sum_{y \in pred(z)} o_y$$
where $pred(z)$ is the non-empty set of preceding neurons to z,
- the biased input $i_z = \hat{\imath}_z + bias_z$,
- the bipolar activation function $h(x) = \frac{2}{1+e^{-x}} - 1$,
- the output $o_z = h(w_z * i_z)$.

Then we call z a standard neuron.

Like in $C - IL^2P$, we represent truth and falsity of a propositional variable by a neuron with an output value in a specified interval. However, instead of using the intervals $[-1, A_{max})$ and $(A_{min}, 1]$ with A_{max} and A_{min} as global parameters, we generalize by parameterizing the minimum false and maximum true output value (-1 and 1 respectively) and by setting all parameters *locally*, that is, for each neuron individually. Let a propositional variable q be represented by a neuron z with output o_z. Then we define:

$$q \Leftrightarrow o_z \in [o_z^+, o_z^{++}] \qquad \neg q \Leftrightarrow o_z \in [o_z^{--}, o_z^-]$$

We will use the term *limits* to refer to the quadruple $(o_z^{--}, o_z^-, o_z^+, o_z^{++})$ of output parameters of a neuron z.

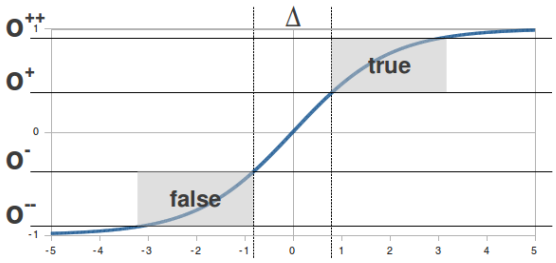


Figure 3.5.: Bipolar Neuron Output and associated Truth Values

Figure 3.5 shows the output of a bipolar neuron together the associated truth values of our algorithm. $\Delta = i_z^+ - i_z^-$ is the difference in activation for truth and falsity and will be of use later.

In order to interpret a neuron output value as a propositional variable, we must ensure that the output intervals for *true* and $false$, are distinct, that is, they do not overlap $o_z^- < o_z^+$. We furthermore constrain propositional truth to positive values and falsity to negative values $o_z^- < 0 < o_z^+$. This will allow us later to define propositional negation as arithmetic negation.

We call a neuron fulfilling this constraint *representative*.

Definition 3.2 (Representative Neuron)**.** Let z be a standard neuron with the output value o_z.

Then we call z representative if the set of possible output values $O_z = \{o_z\}$ is identical to the set of real values $O_z = [o_z^{--}, o_z^-] \cup [o_z^+, o_z^{++}]$ and $o_z^- < 0 < o_z^+$.

With $o_z^{--} \leq o_z^-$ and $o_z^+ \leq o_z^{++}$ the two intervals of a representative neuron z are not empty: $[o_z^{--}, o_z^-] \neq \emptyset \neq [o_z^+, o_z^{++}]$. Along with the output range $[-1, 1]$ of the activation function the following inequalities hold:

$$-1 \leq o_z^{--} \leq o_z^- < 0 < o_z^+ \leq o_z^{++} \leq 1$$

Note that any neuron is representative regardless of its absolute weight: Since from $o_z^- < 0 < o_z^+$ follows $h(w_z * i_z^-) < 0 < h(w_z * i_z^+)$ and $h(x)$ is an odd function ($h(-x) = -h(x)$), a representative neuron remains representative if we change its current weight w_z to another weight w_z' as long as the sign of the weight is the same $sgn(w_z) = sgn(w_z')$. This enables us to set the weight freely without confusing truth values.

Our line of argumentation now goes as follows: Consider the rule

$$q \Leftarrow \bigotimes_{1 \leq i \leq k} p_i \text{ with } \bigotimes \in \{\bigwedge, \bigvee\}$$

We assume that the positive form of all literals p_i has already been translated to the neurons y_i and that these neurons $\{y_i : 0 \leq i \leq k\}$ are representative. Then we create a neuron z that represents q by doing the following:

Output-Input Mapping map the output values o_y of all predecessor neurons y to the input value $\hat{\imath}_z$ such that $\hat{\imath}_z \in [\hat{\imath}_z^+, \hat{\imath}_z^{++}] \Leftrightarrow q$ and $\hat{\imath}_z \in [\hat{\imath}_z^{--}, \hat{\imath}_z^-] \Leftrightarrow \neg q$

Distinct Input increase the weights w_y of all neurons y such that $\hat{\imath}_z^- < \hat{\imath}_z^+$

Representative Neuron set the bias $bias_z$ to a value between $\hat{\imath}_z^-$ and $\hat{\imath}_z^+$ such that z is representative

We obtain a complete representation of the rule without having to impose a constraint on the weight w_z. Instead, we set the weights w_y of the predecessors $y \in pred(z)$. The three steps will be explained subsequently.

Output-Input Mapping

The output-input mapping defines how to derive the input limits of a neuron z from a set of output limits of its predecessors $y \in pred(z)$. These limits determine the input intervals that represent truth and falsity depending on the type of rule (conjunction or disjunction) and the output limits of the preceding neurons.

$$\begin{aligned} \hat{\imath}_z^{++} &= \sum_{y \in pred(z)} o_y^{++} & \hat{\imath}_z^{--} &= \sum_{y \in pred(z)} o_y^{--} \\ \hat{\imath}_{z,and}^{+} &= \sum_{y \in pred(z)} o_y^{+} & \hat{\imath}_{z,and}^{-} &= \hat{\imath}_z^{++} - \min_{y \in pred(z)} o_y^{++} - o_y^{-} \\ \hat{\imath}_{z,or}^{-} &= \sum_{y \in pred(z)} o_y^{-} & \hat{\imath}_{z,or}^{+} &= \hat{\imath}_z^{--} + \min_{y \in pred(z)} o_y^{+} - o_y^{--} \end{aligned} \tag{3.3.1}$$

The limits are calculated according to four scenarios: The maximum truth and minimum false values $\hat{\imath}_z^{++}$ and $\hat{\imath}_z^{--}$ are simply the sum of the corresponding output values and can be considered the *best-case* values as they are the values with the maximum possible distance to the ambiguous value 0.

The *worst-case* values depend on the operator: Conjunctions represent a worst-case truth if their predecessors represent worst-case truth. However, they become false if all predecessors give even the best-case truth value with the exception of one that gives the maximum false (worst-case false) value. The truth values for disjunctions are calculated similarly.

Finally, negation is represented by arithmetical negation. If z represents a rule where the neuron y is a negative antecedent then substitute y by a neuron y' with the negated output limits:

$$\begin{aligned} o_{y'}^{++} &= -o_y^{--} & o_{y'}^{+} &= -o_y^{-} \\ o_{y'}^{--} &= -o_y^{++} & o_{y'}^{-} &= -o_y^{+} \end{aligned} \tag{3.3.2}$$

Since the resulting intervals may overlap, i.e., $\hat{\imath}_z^{-} < \hat{\imath}_z^{+}$ cannot be guaranteed, we "separate" the two intervals by setting the weights w_y in the following step.

Distinct Input

We show that a weight w_y for the predecessors y of a neuron z exists such that the input of z is distinct. We use the parameter $\Delta_z \stackrel{\text{def}}{=} \hat{\imath}^{+} - \hat{\imath}^{-}$ to refer to the difference by which the input intervals of z are distinct. For simplicity, we assume $-i^{-} = i^{+}$, consequently it holds $-o^{-} = o^{+}$. This assumption will be justified in theorem 2.

Theorem 1. Let z be a standard neuron representing a conjunction or disjunction of its predecessors $y \in pred(z)$ as given in equation (3.3.1). Furthermore, let all predecessors be representative standard neurons with $-i^{-} = i^{+}$. Then for an arbitrary $\Delta_z < 2$ the

input of z is distinct by at least Δ_z, i.e., $\hat{\imath}_z^+ \geq \hat{\imath}_z^- + \Delta_z$ if the weight w_y for all neurons y is

$$w_y \geq -\frac{1}{i_{min}^+} ln \frac{2 - \Delta_z}{\Delta_z + 2k} \tag{3.3.3}$$

min refers to the neuron that has the minimum worst-case truth value i_{min}^+ of all neurons $y \in pred(z)$.

Proof. We give the proof for the conjunctive case.

$$\hat{\imath}_z^+ \geq \hat{\imath}_z^- + \Delta_z$$

Applying equation (3.3.1) yields

$$\sum_{y \in pred(z)} o_y^+ \geq \sum_{y \in pred(z)} o_y^{++} - \min_{y \in pred(z)} o_y^{++} - o_y^- + \Delta_z$$

Assuming the worst case for both sides, we have

$$k * \min_{y \in pred(z)} o_y^+ \geq k - \min_{y \in pred(z)} (1 - o_y^-) + \Delta_z$$

which we can simplify to

$$\begin{aligned} k * (o_{min}^+ - 1) &\geq -1 - o_{min}^+ + \Delta_z \\ o_{min}^+ &\geq \frac{\Delta_z - 2}{k + 1} + 1 \\ w_{min} &\geq -\frac{1}{i_{min}^+} ln \frac{2 - \Delta_z}{\Delta_z + 2k} \end{aligned}$$

The proof is similar for the disjunctive case and yields the same result. □

The result is the worst-case lower bound for the weights w_y such that the input of z is distinct by at least Δ_z. In extreme cases, this bound corresponds to the weight bound in $C - IL^2P$. However, our algorithm calculates these parameters locally and benefits thus from any situation that does not represent a worst-case.

The result shows that $\Delta < 2$ is a precondition for all successor neurons, ensuring thereby that Δ is smaller than the maximum possible output difference of a neuron $o^{++} - o^{--} = 2$. Furthermore, i_{min} must be positive if we want positive weights. This, however, holds for all representative neurons, as we will see in the proof for theorem 2.

Representative Neurons

In order to make a neuron representative, we just have to set the bias such that z itself is representative.

Theorem 2. Let z be a standard neuron with distinct input. By setting

$$bias_z = -\frac{\hat{\imath}_z^- + \hat{\imath}_z^+}{2} \tag{3.3.4}$$

z is representative.

Proof.

The input of z is distinct

$$\hat{\imath}_z^- < \hat{\imath}_z^+$$

We add $bias_z$ and obtain the biased input

$$i_z^- = -i_z^+ \text{ with } i_z^+ > 0$$

Since the activation function $h(x)$ is odd and strictly monotonically increasing for $w > 0$, it holds

$$h(w * i_z^-) = -h(w * i_z^+)$$

With $h(w * i_z^{--}) < h(w * i_z^-)$ and $h(w * i_z^+) < h(w * i_z^{++})$, the neuron z is representative according to definition 3.2. □

Note that by setting the bias in between the limits $\hat{\imath}^-$ and $\hat{\imath}^+$, we obtain $-i^- = i^+$ and $-o^- = o^+$.

3.3.2. Algorithm

The final algorithm works as follows. We assume as input a propositional domain theory, that is, a set of rules of the form

$$q \Leftarrow \bigotimes_{1 \le i \le k} p_i \text{ with } \bigotimes \in \{\bigwedge, \bigvee\}$$

We transform a rule r with head q to a neuron z with the following algorithm:

1. If the number k of antecedents in r is 0, then mark z as input neuron and stop.
2. Obtain the neurons y_i by recursively calling this algorithm on all atoms p_i in the body of r.
3. Create a neuron z and connect it to all neurons y_i with weight w_z ($-w_z$) if p_i is a positive (negative) literal.
4. While $\hat{\imath}_z^- + \Delta_z \ge \hat{\imath}_z^+$ do

- Increase w_{y_i} for all $1 \leq i \leq k$.
- Recalculate the output limits for the neurons y_i for all $1 \leq i \leq k$.
- Recalculate $\hat{\imath}_z^-$ and $\hat{\imath}_z^+$ according to equations (3.3.1).

5. Connect z to the bias neuron with weight $w_z * bias_z$ where $bias_z$ is defined as given in equation (3.3.4).

When processing the rule r with head q, the algorithm sets the weights w_{y_i} for predecessors of z such that the input to z becomes distinct (line 4). It then sets the parameter $bias_z$ such that z becomes representative (line 5). Note that we do not set w_z because this will be only set when transforming the heads of rules where q occurs in the body. Since the weight of the output neurons will thus not be set at all, we can simply set it to 1. Note also that the simple search algorithm for weights (implemented as loop in line 4) can be substituted by a more sophisticated binary search on the interval $[1, w_{max}]$ where w_{max} is the theoretical maximum weight as stated in equation (3.3.3)

By using low weights the algorithm is able to deal with the lack of resolution as described in the introductory examples. It remains to show that the algorithm maintains correctness and discuss other properties of the algorithm.

Soundness

Since all input layer neurons are defined such that they are representative and we set the bias of each neuron such that it is representative it holds that every neuron is representative. To show that the transformation of a neuron is sound, we show that the output of a neuron correctly represents a conjunction or disjunction of the variables represented by its predecessor neurons.

Theorem 3. If the neurons y_i represent the variables p_i in the rule $q \Leftarrow \bigotimes_{1 \leq i \leq k} p_i$ with $\bigotimes \in \{\bigwedge, \bigvee\}$ then we can construct a neuron z such that it represents q.

Proof. Due to the monotonicity of the activation function, we limit the proof to border cases.

We show that if all predecessor neurons represent $true$, the conjunction neuron z also represents $true$: $\forall y \in pred(z) : o_y \geq o_y^+ \rightarrow o_z \geq o_z^+$. It follows:

$$\sum_y o_y \geq \sum_y o_y^+ = \hat{\imath}_z^+$$
$$\hat{\imath}_z \geq \hat{\imath}_z^+$$

As all neurons y are representative, we set a positive weight such that $\hat{\imath}_z^+ > \hat{\imath}_z^-$, and from monotonicity of the activation function $h(x)$ follows that $o_z \geq o_z^+$.

If one predecessor neuron $y*$ represents $false$, the conjunction neuron z also represents $false$: $\exists y* \in pred(z) : o_{y*} \leq o_{y*}^- \rightarrow o_z < o_y^-$. The maximum unbiased input $\hat{\imath}_{z,max}$ is

then

$$\hat{i}_{z,max} = \sum_{y \in pred(z)} o_y^{++} - o_{y*}^{++} - o_{y*}^{-}$$
$$\hat{i}_{z,max} \leq \sum_{y \in pred(z)} o_y^{++} - \min_{y \in pred(z)} o_y^{++} - o_y^{-} = \hat{i}_z^{-}$$

Again, we set a positive weight w_y such that $\hat{i}_z^{+} > \hat{i}_z^{-}$, and from monotonicity follows that $o_z \leq o_z^{-}$.

The proof for the disjunction is similar. □

The Problem of Increasing Weights

When transforming a graph to a network, the weights of a node y are processed by all of its successors and might be increased. If a node z sets the weight w_y of one of his predecessors and later a sibling z' of z increases this weight w_y, the absolute output values of y are increased and might decrease the difference between i_z^{-} and i_z^{+} under certain circumstances.

There is no straightforward way of handling this problem due to the setting of w_y and the input limits i_z for each neuron z. The neuron transformation therefore implicitly ranges over two levels of the neural network. The simplest way to handle the scenario is to recalculate the neuron parameters for all previously transformed parent nodes of y and their successors if the weight w_y is increased. This can be time-consuming in the worst-case.

However, knowing that the highest activations occur most likely at the neurons with the highest number of children, we employ a heuristic that transforms the rules with the highest number of antecedents first. This translates to modifying line 2 in the above algorithm such that the atoms are transformed in the (descending) order of the number of their children.

The problem is further relaxed for propositional domain theories with a layered structure (e.g., when in Disjunctive Normal Form). In this case it is possible to transform the domain theory layer by layer. Once a layer is transformed, the weights of the preceding layer need not be recomputed anymore.

Cyclic Domain Theories

A second issue is that, unlike the original $C-IL^2P$, our approach is not directly designed to work on cyclic domain theories such as $a \Leftarrow b, b \Leftarrow a$. Although a scenario relying on both fuzzy evaluation and cyclic domain theories should be less common, cyclic domains do not invalidate our theoretical foundation of a rule-to-neuron transformation.

To deal with such a scenario, the top-level algorithm would have to be modified to avoid infinite recursion. Still, since $C-IL^2P$ provides a solution and our algorithm leads in the worst case to the weights as implied by $C - IL^2P$, our algorithm works as well. Conversely, one might also opt to start from a network constructed by $C - IL^2P$ and gradually reduce weights while neurons remain representative and their input distinct.

3.3.3. Evaluation

While having proved that our algorithm is correct and as such suitable for translating a propositional domain theory to a neural network, we now demonstrate its practical benefits.

Our hypotheses are the following:

Bigger Output Interval Our approach maps states to a bigger subinterval in $[-1, 1]$ than $C - IL^2P$ would.

More States Distinguishable Consequently, our approach is able to distinguish more states from each other.

Increased Playing Strength Consequently, our approach improves playing strength.

We will refer to Nexplayer version using plain $C - IL^2P$ as Nexplayer v1.0 and to the version presented above as v1.1

Bigger Output Interval

Our candidate set of games comprises 190 games. The set is mostly equivalent to the games available on the GGP server [Sch12].

For each of these games, we follow a simple algorithm

- let r be the first role in the game description
- generate all moves possible in the initial state
- execute all moves and for each successor state evaluate the network for role r and the goal value 100

We end up with a set of state values in $[-1, 1]$. The effective output interval is the interval between the minimum and the maximum evaluation of the states at ply 1.

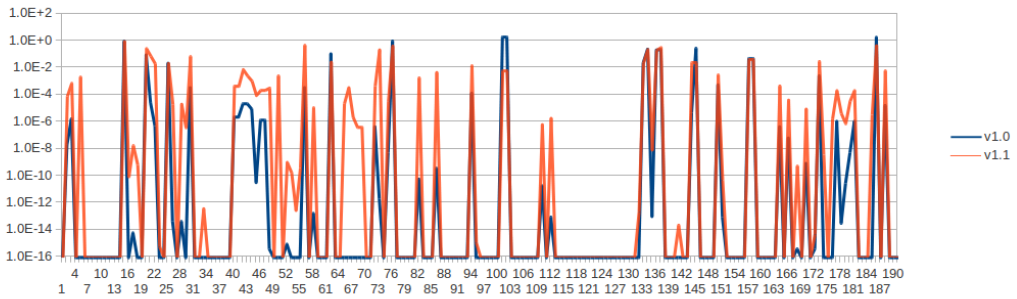


Figure 3.6.: Logarithmic Size of the Output Interval for Ply-1 States of 190 Games

Figure 3.6 shows a graph indicating the size of the output interval for each game. The games are plotted on the x-axis according to their alphabetic order. Since similar games

often have similar names (e.g., Connect Four variants all start with the string "conn"), there are clusters of similar results. The y-axis adheres to a log_{10} scale. We assume the lower bound to be 64-bit floating point precision, that is $2^{-53} \approx 10^{-16}$. We cannot measure any resolution lower than this, so we cap the results at this value.

We can see that the resolution of our v1.1 is higher in 67 games and equal in 117 games to that of v1.0. In six games, the output interval is smaller.

The six games are all games with simple networks with at most three layers, i.e., at most one hidden layer. Here, no resolution problems occur anyway and high weights only push the output values for different states farther away from each other without achieving a higher degree of distinction between states. Figure 3.6 demonstrates this as interval sizes of version 1.0 only exceed those of v1.1 in games where both interval sizes are already big (0.007 or higher).

The 117 games with equal output interval size are mostly games where no atomic part of the goal is affected by the first move. An example is the game `checkers` where the winner is determined in terms of the number of pieces of both players. As it is impossible to remove any stone in the first move, all states are evaluated to the same value.

	Games	v1.0	v1.1
all games	190	$8.8 * 10^{-14}$	$5.8 * 10^{-12}$
games where v1.1 increases the output	67	$9.9 * 10^{-10}$	$2.0 * 10^{-5}$

Table 3.3.: Output Interval Sizes in Games

Table 3.3 shows the average logarithmic output size for all games and for all games where v1.1 increases the output size.

In general, we roughly expect our algorithm to halve the logarithmic output size, that is, increase the size of the output interval of a network to the size of its square root.

More States Distinguishable

To determine the number of distinct states, we work on the same set of games and use the same algorithm as in the prior evaluation of the output interval.

However, this time we count the number of states that obtain a different evaluation from the network. The purpose is to estimate the extent to which the agent is now better informed when moving.

Figure 3.7 shows the number of different states in ply 1 identified by the network for the first role and goal value 100.

We see that in 31 of the cases with a higher resolution, also a higher number of states with distinct state value was identified. Moreover, in none of the cases, the number of distinct states decreased. We can thus expect the agent to make better informed moves in these games.

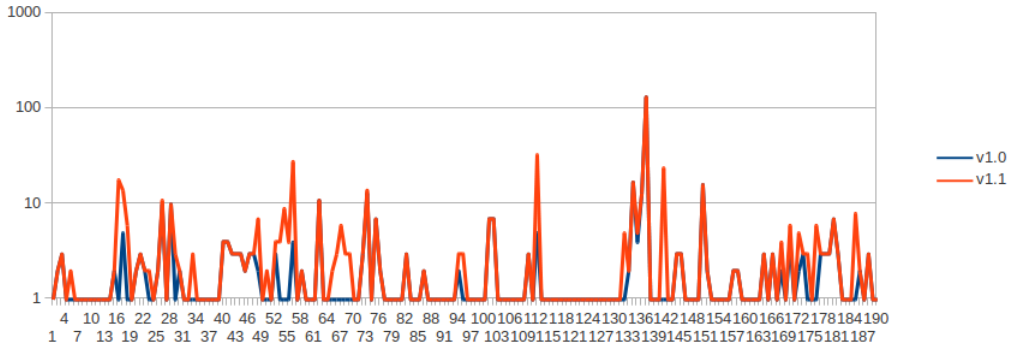


Figure 3.7.: Number of Distinguishable Ply-1 States in 190 Games

Increased Playing Strength

Ultimately, the statistics would be meaningless if there were no impact on the playing strength of the agent. To provide for an easy comparison, we use the same setup as in the evaluation section 3.2.4 of the original neural network without our extension.

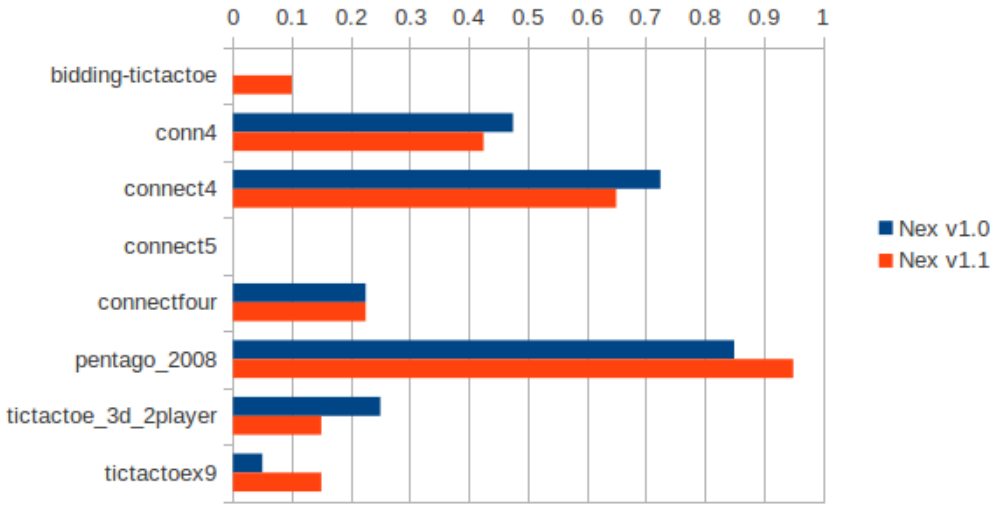


Figure 3.8.: Win Rates of Nexplayer v1.1 and v1.0 against Fluxplayer

Figure 3.8 shows the average points per match obtained in 20 matches with a 40,4 clock setting. As we can see, Fluxplayer plays far better than Nexplayer and achieves in fact almost the same points as against Nexplayer v1.0.

Since the results do not allow for an unambiguous conclusion, we set up 40 matches of Nexplayer v1.1 against Nexplayer v1.0 to check whether there is a difference.

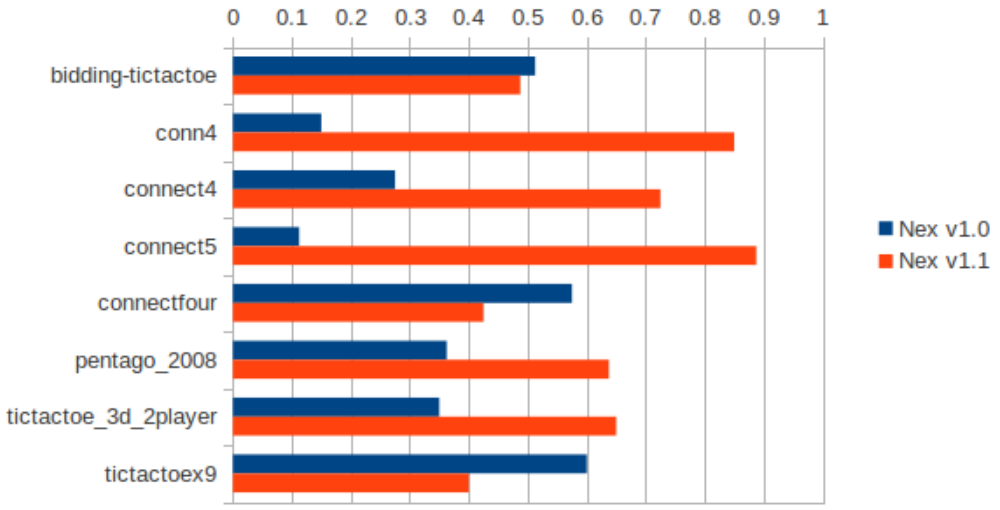


Figure 3.9.: Win Rates of Nexplayer v1.1 against v1.0

Figure 3.9 shows the average points per match obtained in 40 matches with a 40,4 clock setting. We see indeed a huge difference in performance. Nexplayer v1.1 achieves in average 63.3 points compared to 36.7 points for Nexplayer v1.0.

	Fluxplayer	Nexplayer	
Game	-	v1.0	v1.1
Fluxplayer	-	67.8	66.9
Nexplayer v1.0	32.2	-	36.7
Nexplayer v1.1	33.1	63.3	-

Table 3.4.: Nexplayer Versions in Setup against Fluxplayer

For comparison we provide table 3.4 that shows the numeric statistics of Fluxplayer, Nexplayer v1.0 based on the original $C - IL^2P$ and Nexplayer v1.1 based on the new approach.

Further Implications

The evaluation at hand was performed without using any type of features in the evaluation function and on a slightly different set of games than in our workshop paper [Mic11b]. This explains the minor differences in the results.

Results clearly show that in the new approach states are mapped to a bigger interval and that consequently more states are distinguished from each other.

When looking at the effect on playing strength, we get a divided picture. We can see that the high-resolution approach has a very limited effect when playing against

Fluxplayer. The results indicate that there are far more important parts of the evaluation function to be addressed. These parts do not belong of the aggregation function and will be addressed in chapter 4.

Still, in average Nexplayer v1.1 significantly performs better than Nexplayer v1.0, emphasizing the importance of a well-designed aggregation function given an equal base of features (in this case none). Moreover, once we use features our modification to $C - IL^2P$ becomes even more important for our game playing agent: Currently, many states are indistinguishable due to the lack of features, hence no improvements in distinguishing states could be registered. However, with features neurons will not only return the crisp values -1 and 1 but also values in between. Hence, even smaller differences in neuron output values must be recognized and propagated through the network. Consequently, the values obtained should be regarded as a cautious lower bound on the utility of the approach.

3.4. Summary

We analyzed the aggregation functions used by current GGP agents along the lines of efficiency and quality (section 2.5). We found that only deterministic functions can in principal satisfy these criteria.

Still, these have shortcomings with respect to the learning and flexibility that have not yet been addressed in any agent. We showed that neural networks, when initialized with $C - IL^2P$, exhibit all desired properties and are themselves already capable of confronting a modern GGP agent to some degree.

However, due to the hight weights imposed by $C - IL^2P$, the strict monotonicity criterion could not be guaranteed to hold in every game. We reduced the effect of this disadvantage by calculating connection weights locally. Consequently, the playing strength of Nexplayer improved in comparison to its predecessor based on plain $C-IL^2P$

We focused up to now only on small set of games that is especially suited for fuzzy evaluations. In the next chapter we will generalize the set of games by introducing features that allow Nexplayer to play well in other classes of games. Each of these features will enhance Nexplayer in another way, and allow not only for a broader basket of games to be played effectively, but also for performance optimization that boost playing strength in the eight games used for evaluation throughout this chapter.

4. Evaluation Functions II - Features

Features are functions that map the degree of presence (or absence) of a tactical or strategical pattern in a game state to a number. Piece development, king safety, pawn structure and material advantage are well-known concepts in Chess and are therefore encoded into features that form the basis of every deterministic evaluation function.

Features can be considered the elements of an additional abstraction layer that is on top of the plain state information but below the state value itself. The game state representation is evaluated by these features and the aggregation of the feature output forms the state value. Viewing features as part of an abstraction layer also highlights their advantages:

Reduction of Complexity Features operate only on an aspect of a state. Their application allows for a divide and conquer approach by reducing the original state evaluation problem to a smaller subproblem.

Abstraction Features provide a more abstract view of the state because they generalize from concrete state information.

Intuitive Representation Features represent concepts that are intuitive for humans. Therefore they are used as an entry point for tracing the information processing from a state to a value.

This chapter aims to analyze the theoretical foundations, applications and benefits of features in GGP agent systems.

We begin in section 4.1 with a categorization of the different features applied in state of the art GGP agents, organized by their acquisition techniques and the class of games they can be used in.

Section 4.2 generalizes further by addressing the questions of what features estimate and how this should contribute to the evaluation of a state.

We put the results of our analysis into practice by describing an efficient way to detect (section 4.3) and integrate (section 4.4) features into an evaluation function.

We deal with the specific drawbacks of distance features and show how to improve results in section 4.5 and conclude in section 4.6.

4.1. Categorization of Features

The features used throughout game playing agents are very different in terms of acquisition and the games they are applied in. The aim of this section is to put the different

features on a common ground. This will pave the way for an analysis and conclusions on what features serve best our quality and efficiency constraints.

Specifically, we will provide a general view on

- what types of features exist,
- how they can be acquired, and
- how they are used in a GGP agent system.

We conclude by deciding in favor of a subset of features and acquisition methods that will motivate the rest of this chapter.

4.1.1. Methods for Feature Acquisition

While there are many approaches for acquiring features, all of them follow the same pattern:

Feature Construction First, a candidate feature as an algorithmic unit is constructed that operates on a game state and returns a value.

Feature Selection Among all candidate features those are selected that, while being expressive, are not too expensive in computation.

Feature Integration All selected features are integrated into the evaluation function.

There is a number of terms used throughout the scientific community to refer to one or all of the above three acquisition steps. Most common among these are feature construction, feature discovery, feature extraction and feature generation. However, each of these is ambiguous as it either uses the name of a single step to describe the complete process (discovery) or suggests that a specific method was used (construction, extraction, generation). To avoid these ambiguities, we will refer to the overall process as feature acquisition, highlighting the result rather then the means of the process.

In the following, we discuss each step with the term as given above.

Feature Construction

Feature construction is the first step of feature acquisition and defines the set of candidate features that can potentially be used in the evaluation function.

An optimal feature construction algorithm is characterized by

Completeness The set of candidate features it constructs contains *all* features that an optimal evaluation function would use.

Specificity The set of candidate features it constructs contains *only* features that an optimal evaluation function would use.

An incomplete construction algorithm constrains the evaluation function by not providing some good features, thereby decreasing evaluation quality. An unspecific construction algorithm creates more candidate features than necessary. Since candidate features are then passed on to the selection and integration step, an unspecific construction reduces the efficiency of evaluation function construction. Moreover, if selection and integration are imperfect, it also reduces evaluation quality.

Candidate features can be obtained in several ways:

Definition They can be predefined by the agent programmer as done in Metagamer [Pel93].

Generation They may be generated synthetically by connecting elements of a game state with connectives. For example, one could randomly connect fluent propositions with logical connectives.

Detection They can be detected as an expression in the game rules that can be transformed to a feature. The transformation is what Clune referred to as imposing an interpretation on an expression [Clu07]

Derivation They can be derived from already existing features as done in Zenith [Faw93].

We will analyze the four ways of obtaining features along the lines of completeness and specificity and their implications on evaluation function quality and efficiency.

An example for *predefined features* are mobility and piece value features. The mere fact that feature acquisition is an important topic discussed in every publication on deterministic GGP agents implies that there are no predefined features that generally serve well. Most often they are used as a complement to other construction algorithms.

An approach for synthetic *feature generation* would be to modify the language of propositional logic formulas where the set of propositional variables is defined as the set of ground fluents of a game. Each word in the resulting language can then be considered a (boolean) feature of that game. The problem of such a generative way of feature construction is that the space of generatable features is infinite, and, even if we allow only the smallest formulas with a unique meaning, the number of generatable features is exponential to the number of ground predicates that in turn is potentially exponential to the number of variables of non-ground predicates. Given that we need to evaluate every feature for the selection and integration step while only obtaining propositional features with a crisp output, the results are not worth the effort.

Put more generally, the generative component of feature construction represents an uninformed search in the (possibly restricted) feature space. The features generated are thus unlikely to be expressive but still require time-consuming selection and integration. Consequently, to date there is no GGP agent system applying synthetic feature generation.

There are, however, mixed approaches that apply some sort of feature generation. For example, Kuhlplayer[KDS06, Kuh10] first tries to detect game elements such as boards and their content (markers and pieces) and then generates a number of features (e.g., distances between markers, quantities of markers) that operate on these game elements,

regardless of whether they make sense in the game or not. One could argue that this approach follows the feature generation pattern for a restricted subclass of features.

Feature detection avoids the problems of blind search in the feature space since it operates on preselected expressions. For instance, Cluneplayer[Clu07] uses expressions occurring in the goal rules as starting point for features. This restricts the size of the generated set of features. Naturally, these expressions are likely to be useful for state evaluation since they occur in the goal rules. Cluneplayer transforms them to features by imposing up to three interpretations on these expressions. These features can be expected to "inherit" the usefulness from the expressions they are build from. Thus, though not guaranteed, the features detected this way are likely be complete and specific.

Finally, *feature derivation* is used to a small extent in Cluneplayer where relative features are derived from symmetric absolute features. For instance, instead of using two features that return the number of pieces for a role in Checkers, the agent uses the difference between both piece counts as an arguably more expressive feature.

A more general approach on transforming features is discussed in [Faw93] and there is a follow-up work based on GGP [Gün08]. Both works are able to produce a variety of good features. The main problem for both is that the number of candidate features generated is too big, leading to the same problem as feature generation, namely prohibitively high time requirements for selection and integration.

Feature Selection

Feature selection is the process of selecting which of the candidate features obtained during the construction phase will actually be used in the evaluation function. Since computation time for state evaluation is limited, a feature's utility is determined by the quality it contributes to the state evaluation relative to the average time it requires to be evaluated. Thus, the main criteria for features are run time and quality.

Run time is typically considered by eliminating candidate features whose average run time cost exceeds the average feature evaluation cost by a predefined margin. The reason is that the least efficient feature sets the run time complexity of the complete evaluation function.

Quality on the other hand is not directly measurable, however, other properties are used in literature that relate to feature quality. These are:

Stability More stable features are more valuable as they indicate long-term state information. Likewise, features with wildly oscillating outputs are dangerous since they represent short-term properties lead to short-sighted play. Cluneplayer uses stability as main criterion.

Correlation with Goal A strong positive or negative correlation with the goal value of a state indicates utility of a feature. All GGP agents with the exception of Fluxplayer[ST07, Sch11] rely on correlation.

Inheritance A feature that is derived of other useful expressions or features is likely to be useful as well. Cluneplayer and Fluxplayer justify features with inheritance.

Stability is used explicitly only by Cluneplayer[Clu07]. Cluneplayer considers an expression stable if it varies little from one state to its successor, relative to the variations from the start to the terminal state. It uses a simulation approach for determining the stability value of the feature, that is, it simulates random matches and returns the average stability determined in these matches as value. Naturally, this puts the reliability of the value in a direct trade-off with the run time costs and is therefore suboptimal. Moreover, random matches can be quite different from matches played by informed players, so the value obtained is not necessarily good, even for a high number of random matches.

Correlation with the goal value is a much more popular measure of quality and is used by almost all GGP agents. Much like stability, a simulation approach is used for determining the correlation, leading to the same suboptimal trade-off between run time and reliability. Even worse, goal values are less likely to be representative than stability values as they are much more sensitive to errors induced by random play, decreasing further the reliability of any sampled correlation value. Also, for each random match we only obtain exactly one goal value, increasing the costs of determining correlation. Cluneplayer handles the excessive run time costs using pseudo-terminal states. These are states obtained by constructing terminal states from terms. Obviously, this works only in some games, as can be seen when asking an agent to construct a check mate state. In these cases, however, it eliminates the need for internal simulation for finding terminal states at the risk of obtaining correlation values using unreachable states.

Finally, *inheritance* is strongly related to feature detection and derivation. If the feature is detected based on an expression used in the goal function of the game, importance (i.e., correlation with the goal value) can be considered inherited from the expression. This approach is used by Cluneplayer and Fluxplayer. On the other hand, if the feature is derived from another feature, the correlation is not necessarily ensured and a further proof of utility may be required. Still, it is more likely to be useful than a randomly generated feature. Cluneplayer uses this form of inheritance to a small extent when combining two absolute features into one relative feature.

Feature Integration

The candidate features that are not removed during feature selection are used in the evaluation function.

For this purpose, the feature has to pass through

Normalization The feature output needs to be normalized to fit in the aggregation function.

Weighting A weight of importance and direction must be assigned

Placement The argument position in a (possibly nested) aggregation function has to be determined.

Normalization is needed because features have different output intervals and the meaning of a feature usually depends on the output relative to its output interval. For in-

	Efficiency	Quality
Construction		
Definition	+	-
Generation	-	-
Detection	+	+
Derivation	(-)	(+)
Selection		
Stability	-	-
Correlation	(-)	-
Inheritance	(+)	(+)

Table 4.1.: Summary of Properties of Feature Acquisition Techniques

stance, 3 stones in Nine Men's Morris is the lowest possible value for a feature that counts stones, 9 is the highest. It is therefore better to design a feature with an output interval $[0, 6]$. than one with an interval $[3, 9]$. In the same way, normalization may require a non-linear feature output in terms of the game concept. For instance, losing one stone in Nine Men's Morris when you still have 9 reduces the state quality much less than losing a stone when you only have 4. Expressing such subtleties may prove useful.

Normalization is mandatory if the aggregation function provides a context that interprets the feature output, as is the case in fuzzy aggregation functions.

Weighting assigns weights to the features found. These weights are usually determined using the quality measures determined in the selection phase to avoid additional simulations. In the case of a linear combination as aggregation function, feature weighting and normalization are often performed in one single step. Fluxplayer is the only agent that performs no feature weighting[ST07].

Finally, *placement* is trivial in linear combinations due to the commutative nature of the aggregation function. Fluxplayer with its non-commutative aggregation function places features according to the placement of the underlying feature expression in the goal function.

Summary

An overview over the quality and efficiency properties of the feature construction and selection techniques is show in table 4.1. Efficiency here refers to the efficiency with respect to constructing and selecting the feature (as opposed to evaluating it).

Feature construction is only efficient if the features are defined beforehand or detected on the restricted set of expressions. Of these two, only feature detection is likely to produce a comprehensive set of features.

On the selection side, stability and correlation suffer from increased time requirements as they require simulations and/or terminal states to be determined. Still, the values obtained are not guaranteed to be useful due to being result of random playouts.

The feature integration steps do not incur significant run time costs and also involve

no decision that affects the quality of the evaluation function. Instead, they are a mere consequence of the aggregation function selected and the construction and selection algorithm employed. Therefore we omitted them in the table.

As an intermediate result of our analysis we conclude that features detected in the game rules and justified by inheriting their usefulness from an underlying expression should provide the best results in terms of quality and efficiency.

4.1.2. Feature Types

Based on the GGP agents, we can categorize features into two categories:

Class-based Features are (pre)defined features applicable only in a class of games.

Rule-derived Features are derived from expressions in the game rules

All features used by GGP agents belong to one of the types with some being a mixture of the two.

Class-based Features

Class-based features are features applicable only in a class of games. Feature construction therefore depends on whether the game at hand can be identified as a member of a class of games. Once a class could be identified, features that are usually useful in this class are included as candidate features. Since it is unclear whether the feature actually is useful or not, the feature has to go through the usual selection and integration step.

General Games The most general class-based features are based on properties that are fulfilled in every general game. Features of this class are always applicable and can thus be algorithmically prepared. While allowing for an application in every game, they are very unspecific to be of any outstanding value. Besides, it is unclear whether they have any effect on the state quality and how to integrate them best in that case, resulting in the need for a further selection, e.g., to determine weights. The following is a list of common features for general games, all three are applied, for instance, in the GGP agent Ogre [Kai07a].

Mobility [Sha50] reflects the number of actions available in a state. As by definition in every non-terminal state there is a legal move for every role, mobility is always applicable and a prior domain analysis is not required. Though it is hard to guess from the rules whether the impact of mobility on the state utility is positive, i.e., whether a high number of possible actions for a role is favorable for the role, mobility indicates one slight advantage: the right of choice. The player with just one action to perform has no influence on the state and the more actions are available, the more parts of the game tree are usually accessible.

Depth returns a value that represents the depth of the state in the game tree. The feature can be used to discount the value of states deeper in the game tree. As a consequence, the evaluation function prefers states higher in the game tree, that is, states with shorter solution paths.

Finally, *Goal Value* is another simple feature applicable to the set of games with known goal definition. The goal value feature evaluates a state using the goal function of the game. Though the goal function is required to be descriptive only on the set of terminal states, often it already yields useful information during a match. An example would be a game where a role gains a predefined number of points for each opponent piece captured. If the number of captured pieces per role is stored in a fluent and the goal function only refers to that fluent, we have a defined goal function in all states. The games `brawl` and `skirmish` are such games. On the other hand, if the goal function only yields descriptive results in terminal states, its value is constant during the game and thus does not affect the evaluation of a state. This usually happens for goal functions that implicitly capture the terminal condition. So in any case the goal value is likely to improve the evaluation, or at least, not distort it.

Board Games A more specific class is the class of board games, relevant, e.g., for Kuhlplayer[KDS06]. Boards can be represented as a graph where the nodes represent locations. On these locations game elements such as mobile pieces or stationary markers can be placed. Typically, boards are assumed to have a one-to-one relation between a non-empty board location (cell) and the board element that occupies it (content): This means that for each location on the board there is at most one board element (content), i.e., no two board elements can share the same location. Likewise, no board element can occupy more than one location.

Naturally, we can determine *distances* between board locations by determining the length of shortest paths in the underlying graph. Moreover, knowing that each board element has (a set of) associated coordinates, we can also determine distances between a board cell and a board element or between two board elements. Besides, quantities of board elements can be determined.

Board Games with Pieces A subclass of board games are board games with *pieces*. In addition to detecting a board, this requires to detect board elements that are movable. This is the distinguishing property that separates board games like Tic-Tac-Toe or Connect Four from Chess-like games. In case such movable board elements can be detected, we can apply additional features that are derived from the mobility (number of actions) of each piece.

Piece value is the primary feature and correlates to the upper bound on mobility. Here, each board element, called piece in this context, is assigned a value (representing the average or maximum mobility of the board element) and the sum of the number of board elements weighted by their piece value is the overall piece value of the state. A high overall piece value indicates that a high mobility can potentially be achieved and moderates the effect of spikes and dips in mobility.

Other features are either also directly derived from mobility, for example x-ray mobility, that measures the mobility of a piece after removing all movable obstacles (other pieces) on the board that constrain its mobility. Or they are derived from a piece value such as the threat feature, that measures how many piece value points of the opponent

can be instantly removed by capturing. While it may be worthwhile to specialize a game playing agent towards this class of games, our primary interest lies in the detection and construction of general and adaptive features. Therefore we refer the interested reader to Metagamer [Pel93], which is the primary source for this type of features. Since most of the mentioned features are an integral part of Chess heuristics, we recommend the heuristics of GNU Chess [JS91] as a short but instructive example on features for board games with pieces.

In summary, all class-based features share the same properties, namely that they are general within the class of games and can be applied once the game class is confirmed. In addition, features based on more specific game classes are typically more descriptive than more general features. With respect to efficiency, game class detection may be expensive but needs to be done only once to unlock a number of features. Efficiency is thus in a trade-off with the features that could potentially be applied.

While there is no prohibitive disadvantage related to class-based features, their application in a GGP agent can only improve an existing heuristics in the given game classes but not provide a good heuristics in general games on its own.

Rule-derived Features

Rule-derived features are directly derived from an expression in the game rules. If the expression is in some way relevant in the game, then the feature is most likely relevant for state evaluation as well. Therefore, rule-derived features are more likely to produce expressive features.

The two agents mainly using rule-derived features are Cluneplayer and Fluxplayer. For feature detection, Cluneplayer relies on three interpretations (solution cardinality, partial solutions, distance) that are imposed on expressions occurring in the goal condition. For each applicable interpretation on such an expression a candidate feature is obtained.

Fluxplayer can be seen to apply the same interpretations, although in a very different way: The solution cardinality and partial solution interpretations are implicitly realized using a fuzzy logic aggregation function. Evaluations of conjunctions here represent partial solutions and evaluations of disjunctions solution cardinality. Distances are discovered in a similar way as in Cluneplayer.

For selection and integration, both agents differ: Cluneplayer determines for each feature its stability and its correlation to the goal value based on samples. These values are then used for both selecting good features and weighting the features in a linear combination. Fluxplayer, on the other hand, substitutes evaluations of expressions in the aggregation function by features derived from the substituted expression. Thus, the fuzzy aggregation function embeds the feature and provides a context for integrating it. This means that while the feature must be normalized to the interval of fuzzy values, there is no necessity to determine a weight. Therefore, a costly qualitative evaluation of a feature (determining stability, correlation and weights) can be skipped.

4.1.3. Conclusions

With respect to the criteria of an evaluation function as given in section 2.5, we are interested in expressive features that can be acquired and evaluated quickly. The rule-derived features detected by interpreting expressions as done in Cluneplayer seem reasonable and most general.

However, given the feature acquisition techniques, it is mostly the need for determining correlation, stability and weights that incur high run time costs. We can avoid this by using rule-derived features based on these expressions occurring in the fuzzy aggregation function as done in Fluxplayer.

In this way, we combine the generality of Cluneplayer with the efficiency of Fluxplayer.

Since rule-derived and class-based features are not mutually exclusive, we can still decide at a later point whether and how we want to include class-based features.

4.2. A New View on Features

Before presenting our approach on rule-derived feature acquisition in our own GGP agent, we want to take a closer look as to what distinguishes a rule-derived feature from a simple evaluation of the expression it is derived from.

In the following we assume a game description D and the corresponding set E defined as the set of valid GDL formula in D. We will extend this definition to the binary predicates $equal/2$, $greater/2$, and $smaller/2$ that use natural numbers as arguments and have the same meaning as their arithmetic counterparts $=, >, <$, respectively. We also remind the reader that the predicate $distinct/2$ is predefined and has a meaning equivalent to the operator $\neq$ on natural numbers.

Also, we earlier defined the macro holds (see section 2.1.4) on an expression ϵ that is true in a state if the expression ϵ holds in that state.

We define a real variant of the macro as follows:

$$holds_R(\epsilon, s) \stackrel{\text{def}}{=} \begin{cases} 1 & holds(\epsilon, s) \\ 0 & else \end{cases}$$

We define a feature ϕ derived from expression ϵ so that it does essentially the same as $holds_R$ but maps to the real unit interval $[0, 1]$ instead of $\{0, 1\}$.

$$\phi(\epsilon, s) \rightarrow [0, 1]$$

This definition agrees with our understanding of a feature as mapping a state to a value. The only difference is that we additionally constrain the output value to the interval $[0, 1]$. Obviously, given an expression ϵ, $holds_R$ and therefore $holds$ can be seen as a special case of feature ϕ.

Current View

In order to be of additional use, the feature ϕ must incorporate more information than a boolean $holds$. The corresponding questions is: What is the additional information contained in the feature ϕ.

The answer expressed in the current GGP agents is that the additional value is the discounted value of the expression in future states. We can observe this paradigm in many agents:

Fluxplayer [ST07] uses distance heuristics to obtain information how a fluent may evolve in future states. Consider an expression $true(step(X)) \wedge equal(X, 1)$ and two states in which X takes the values 7 and 2. So, while the expressions $equal(7, 1)$ and $equal(2, 1)$ are both false, it is far more likely that the latter may hold in the next state given ordinary game mechanics. To reflect this future possibility, Fluxplayer assigns a higher value for the latter expression. Order heuristics are similar, only the underlying expressions would now use $distinct(7, 1)$ or more often $greater(7, 1)$ instead of $equal(7, 1)$.

Interestingly, even the aggregation function of Fluxplayer itself is a feature with a future perspective: It is much more likely that a conjunction holds in the future state if some of its conjuncts hold in the current state. Thus, while both evaluate to $false$ in the current state, the "more true" conjunction deserves a higher evaluation than the "less true" conjunction.

Cluneplayer [Clu07] expresses a similar perspective on features, most obviously in the stability criterion for features. However, even the three interpretations (solution cardinality, partial solution, symbol distance) are mere fuzzified expression evaluations for disjunctions, conjunctions and atoms, respectively.

Even general features used across all agents show such a future perspective. For instance, a piece value feature represents a future estimation of mobility and is generally assumed to be more useful (if applicable) than plain mobility [Sha50].

Naturally, each feature assumes silently a state inertia, that is, the representation of the state only changes little from one state to the next.

4.2.1. A New Perspective

While the current view is certainly justified on a empirical basis, it is neither an explicit theory with formally accessible parts nor does it allow for predictions of the quality of features.

Our goal is not to establish a complete theory on this matter either, however, we can provide insights on how the current view is insufficient in many cases and how a modern feature theory should look like.

In our view, the central information of the evaluation of a feature derived from an expression in a state s is the expression value in the terminal state reached from s.

To formalize this view, we define the function $T(s)$ as returning the set of all terminal states reachable from s. An oracle function $\tau(s)$ takes a state s and returns the terminal state $t \in T(s)$ that is eventually reached.

The we define the perfect feature $\phi^*(\epsilon, s)$ as:

$$\phi^*(\epsilon, s) = holds_R(\epsilon, t) \text{ where}$$
$$t = \tau(s)$$

Naturally, this definition is of little practical use given that in most games we have no means to obtain the oracle τ: While exhaustive search is usually not an option due

to time constraints, a move selection or state evaluation oracle (see section 2.3) are not available which is why we discuss feature acquisition in the first place.

Anyhow, the formula already expresses a fundamental property of features: The evaluation of an expression in the current state is only relevant to the extent to which this affects the value of the expression in the terminal state.

To support this hypothesis, we will briefly view the two extreme cases.

In one case, an expression holds in the current state, but its value is a randomly determined every round. Then the expression evaluation of the terminal state that we eventually reach is independent of the expression evaluation in the current state. In other words, the current expression evaluation is irrelevant.

In the other extreme, an expression holds in the current state and persists through the rest of the match. Then the expression also holds in the terminal state and therefore the current evaluation is relevant.

Since there is no reason not to assume monotonicity between the two extreme cases, we generalize this view to our hypothesis.

Features

In the absence of a terminal state oracle, a the construction of a perfect feature is difficult. To deal with this fact, we must discard the oracle and work instead with the set of reachable terminal states. Since in each of these terminal states the expression may hold or not, we can model the feature value as the probability that the expression ϵ holds given the current state s. We use the feature ϕ as an approximation to the perfect feature ϕ^* and define it as:

$$\phi(\epsilon, s) = P(holds_R(\epsilon, t) | t \in T(s))$$

To build on this view we need to determine, sample or estimate the distribution of terminal states with respect to a specific expression.

Most often, it is impossible to determine this distribution as it would require to build up the game tree in order to count the occurrences of states where the expression holds. Again, it is for the very purpose of not having to build up the game tree that we use deterministic evaluation functions. We will later discuss the concept of persistence which is the only notable exception to the rule (section 4.3.4).

4.2.2. Sampling Features

Sampling the distribution is straightforward as Monte Carlo-based agents demonstrate. Equation (4.2.1) is instantiated such that ϵ is the goal condition and $P(t = \tau(s))$ can be directly sampled.

$$\phi(\epsilon, s) = \sum_{t \in T(s)} holds(\epsilon, t) * P(t = \tau(s))$$

The obtained Monte Carlo state value is thus literally the aggregated average of the state values of the sampled states and can be considered a special case of our view on features.

Note that $P(t = \tau(s))$ is equally distributed among all paths to terminal states, highlighting one of the major problems of Monte Carlo approaches.

Sampling more fine-grained expressions than the goal formula seems an interesting approach, however, as to our knowledge this has not been attempted yet.

4.2.3. Estimating Features

Finally, we can estimate the distribution heuristically.

Assuming an expression $\epsilon = inst(X) \wedge distinct(X, 1)$ where X is first instantiated in $inst(X)$ and then evaluated as to whether it is distinct to 1, we do not know the probability that it holds in the terminal state given that $X = 3$ in the current state. However, if states do not radically change, we can assume that

$$P(holds(\epsilon, t)|holds(inst(3), s)) > P(holds(\epsilon, t)|holds(inst(2), s))$$

Likewise, if for an expression $\epsilon = line(x)$ we know that

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

then we can assume

$$P(holds(line(x), t)|holds(true(cell(1, 1, x)) \wedge true(cell(2, 1, x)), s)) > P(holds(line(x), t)|holds(true(cell(1, 1, x)), s))$$

4.2.4. Conclusions

There are various consequences that arise from our proposed view on features.

Expression Evaluation

If a heuristic evaluation function is to be more than a mere fuzzy evaluation of the state, we need to be aware of the evolution patterns of expressions.

If the expression evolves slowly (i.e., it is stable, its evaluation is not expected to change in the near term) its evaluation to a real number should be higher than if it evolves (changes) rapidly. Real evaluations of expressions, according to our proposed view, depend on the probabilities of an expression maintaining or changing its boolean evaluation. Thus, *expression evaluations vary across expressions*. Therefore, two expressions with the same boolean evaluation do not imply the same real evaluation.

$$holds(\epsilon_1, s) = holds(\epsilon_2, s) \not\Rightarrow \phi^*(\epsilon, s) = \phi^*(\epsilon, s)$$

Thus, e.g., the decision to assign constant values p $(1-p)$ to true (false) fluent evaluations in Fluxplayer is not correct.

The same principle holds along the temporal dimension: *The same expression evaluation in two different states does not imply the same feature evaluation.*

$$holds(\epsilon, s_1) = holds(\epsilon, s_2) \not\Rightarrow \phi^*(\epsilon, s_1) = \phi^*(\epsilon, s_2)$$

The reason is the difference in the number of states that we pass through until we reach a terminal state. An intuitive example is the evaluation of a highly unstable expression. If we pass through dozens of states, we can expect the value of the evaluation in the terminal state to be independent of the current evaluation, however, if the current state is already terminal, the current evaluation is highly stable, despite its intrinsic instability. Instead, we must assign a higher real evaluation (for true expressions) for the closer we are to the terminal state.

Note how this feature combines nicely with the extrapolation property of section 2.5.

Another property is that *features are not symmetric* with respect to the evaluation of the underlying expression. Assuming two states s_1 and $s_2 = s_1 \cup s_\Delta$, this means

$$\nexists f : \phi^*(\neg\epsilon, s_1) = f(\phi^*(\epsilon, s_2))$$

Thus, there is no general way to "negate a feature value" based on the expression value since the corresponding set of reachable states $T(s_1)$ is potentially unrelated to the set of reachable terminal states $T(s_2)$.

In other words, from knowing how a feature evaluates an expression ϵ that holds we cannot infer the feature evaluation if the expression does not hold. Using the term "stability" as an expression property is thus incorrect since only evaluations of expressions can be stable.

Game and Opponent Analysis

While many games today can be played well by specialized game playing agents there are two instances of deterministic games with complete information that pose or posed a problem to the AI community.

The first is Othello where the plain goal to possess more stones than the opponent was found to be a bad heuristics [Bur02]. Our theory would have predicted the same result since the color of a stone is highly instable and thus possesses little predictive value.

$$P(holds(true(cell(5,5,red)),t)|holds(true(cell(5,5,red)),s)) \approx$$
$$P(holds(\neg true(cell(5,5,red)),t)|holds(true(cell(5,5,red)),s))$$

The only exception to this are the border and the corner stones that are much more stable since there are few (corner stones: zero) ways to turn them. Our view on features would thus have predicted the problems of heuristics in Othello.

The second game is Go [GS07] which has a similar goal and the same problem of a bad heuristics. In Go, however, stones do not switch colors and can at most be

removed. We believe that Go is a problem because stones have very different stability values. The problem is that if a structure in Go is unstable (i.e., it can be removed) its neighbors may become unstable as well. Thus the evaluation of the stability of a stone recursively depends on the stability of its neighbors. This broadens the scope of the stability evaluation of a stone to a whole area and allows errors in stability evaluation to propagate and possibly reinforce themselves. Even worse, since a stone is the neighbor of its neighbor, the problem contains itself as a subproblem.

In both games a trivial deterministic evaluation based on the goal condition provides a weak evaluation function. The problem in Othello is the unknown stability parameters of expressions and could thus be resolved. The problem in the second is the interdependence of expressions that cannot be captured with a standard aggregation function. We suggest thus the following principles with respect to games:

- A high playing strength of standard deterministic aggregation functions can be easier achieved in games with weakly interdependent expressions.
- Monte Carlo-based agents are better at games with highly interdependent expressions due to representing this interdependence via simulation of game mechanics.
- Aggregation functions with recursive elements such as recursive neural networks may be able to represent such interdependencies.

Besides, we are currently undecided as to a (cor-)relation of the stability of an expression to the playing strength of the opponent. We expect strong opponents to stabilize features with a favorable evaluation and destabilize other features. We thus have another way of either identifying opponent strength given feature evaluations, or evaluating states given opponent strength.

Future Work

The topic of interpretation and evaluation of features is vast and cannot be covered in detail in a thesis on General Game Playing. We expect to have provided some concrete insights into *what features evaluate* and *how this is best done*. Nevertheless, a detailed theory is clearly lacking and is an outstanding point for further advance.

Since our view is strongly based on probabilities, we also propose to start looking for a deterministic aggregation function based on probability theory. While our current aggregation function is compatible with the probabilistic view on features, there might be more natural ways to deal with the problem. Note how our proposed theoretical framework already supports probabilistic evaluation of expressions. An extension of the $holds_R$ macro to the real unit interval $[0, 1]$ (currently $\{0, 1\}$) would thus allow for an application with states that can be modeled probabilistically such as partially observable states.

In practical terms, we believe there is a lot of room for calculating weights and stabilities in heuristic functions, a subject on which little time was spent in Game Playing. Specifically we believe that, given expressions with weak dependencies between each other, transformation graphs can be build that relate the possible transformation of

expression evaluations to each other. Using such an transformation graph, we could determine expression stabilities and thus parameters of perfect features.

Likewise, we may reduce the scope of Monte Carlo methods to smaller and more fine-grained expressions instead of the complete goal condition. The behavior of smaller expression might be simulated separately in such a transformation graph and would allow for the abandonment of the expensive simulation part altogether if the goal condition can be completely decomposed to such expressions.

Both practical suggestions address the same problem from a different starting point and suggest a convergence of probabilistic and deterministic evaluation functions.

4.3. Detection of Rule-Derived Features

For the purpose of feature detection, we propose a general set of features based on expressions.

We start by defining the input and output of the feature detection algorithm, that is, what expressions are accepted and what is its expected output. We proceed by showing how to detect solution cardinality, distance and persistence features. In addition to syntactic conditions, distance features must fulfill some other conditions that form the basis of their evaluation later and will be discussed next. We finish this section with a summary of the relation between an expression and the stability of its evaluation represented by the feature.

Note that we will only discuss the detection part of each feature. Specifically, we will not address the integration of a feature or show how to evaluate it. These subjects will be discussed in section 4.4.

Instead, we will motivate the feature, specifically with respect to our view on features as given in section 4.2. For this purpose, we express a relation between an expression ϵ holding in the terminal state and a variable measured by the feature in the form of a statement about probabilities, such as

$$P(holds(\epsilon, t)| \ldots) \sim x$$

Here x is a measurement function on the state that will be needed later for feature evaluation and the symbol $\sim$ is to be read as "correlates to". Besides, we will discuss the conditions which must be fulfilled in order to use the feature on expression ϵ.

The content of this section is derived from our workshop contribution "Heuristic Interpretation of Predicate Logic Expressions in General Game Playing" to the GIGA Workshop 2011 [Mic11a].

4.3.1. Input and Output

We consider feature detection an algorithm that, given a game description D, receives an expression ϵ and returns a set of features. For our purposes, the expression will be always a valid GDL formula and we will use both terms, expression and formula, interchangeably.

We can further restrict the eligible formulas for feature detection to be a conjunction of the form $b_1 \wedge \ldots \wedge b_n$ where there is at least one variable X (existentially) bound over all literals b_i for $1 \leq i \leq n$.

This assumption is not only compatible with the definition of a GDL clause as a logic program clause but also is sufficient to represent the information we are interested in – ground literals and conjunctions with variables bound over them.

All other formulas can be decomposed further:

Disjunctions

$$\exists X : (p(X) \vee q(X)) \Leftrightarrow \exists X : p(X) \vee \exists X : q(X)$$

Negations of the form $\neg(\exists X : p(X))$ of are disallowed by GDL [LHH+08] since each variable occurring in negative literal must occur in the head or another positive literal in the body.

Conjunctions

$$\exists X : (p(X) \wedge \exists Y : q(Y)) \Leftrightarrow \exists X : p(X) \wedge \exists Y : q(Y)$$

In order to address the different atoms that may form an expression ϵ we will use the following terms:

Ground Fluent Queries have the form $true(f)$ where f is a ground fluent

Non-ground Fluent Queries have the form $true(f)$ where f contains at least one variable

Recursive Predicates refer to predicates p where there is a (conjunction) clause c in the game rules and p occurs in the head and in the body of the clause

Our general feature detection algorithm accepts a fluent query or a conjunction with variables and returns a set of features that it was able to detect. Since each feature depends on different conditions, we cannot give a generalized description or formulation of the requirements. However, section 4.3.6 will summarize the most important aspects of feature detection.

4.3.2. Solution Cardinality Features

Solution cardinality features try to relate the number of solutions of an expression with variables to the stability of its evaluation. Both, solution cardinality and partial solution cardinality are used by Cluneplayer[Clu07] and represent a form of fuzzy evaluation of disjunctions and conjunctions in a non-fuzzy aggregation function.

Correspondingly, their occurrence in fuzzy aggregation functions is much less frequent. Still they are useful since the fuzzy aggregation only works on ground-instantiated expressions which is not always possible.

In addition, we can apply solution cardinality in cases in which a ground-instantiation and fuzzy evaluation would not provide more information than a solution cardinality feature. This is, for instance, the case for non-ground fluent queries. In this way, we keep the evaluation function (i.e., the network size in Nexplayer) small and evaluation efficient due to the overhead avoided.

Solution Cardinality

Solution cardinality can be applied on any expression with variables. As a minor restriction, we can report that we did not yet see an example of a recursive predicate with variables where solution cardinality would be a sensible feature.

An example is the following expression occurring in the game Tic-Tac-Toe.

```
1 true(cell(X, Y, b)).
```

The idea of the feature is that the more solution to the expression exist, the higher the probability that the expression still holds in future states. Naturally, we assume that the number of solutions from one state to another does not change rapidly.

Assuming $n(\epsilon, s)$ be the number of solutions to an expression ϵ in a state s, we can state that the stability of a true evaluation of ϵ correlates to the number of its solutions:

$$P(holds(\epsilon, t)|t \in T(s) \wedge holds(\epsilon, s)) \sim n(\epsilon, s)$$

The solution cardinality feature is comparable to a disjunction that would be built if we follow an approach of ground-instantiation.

Partial Solution Cardinality

Partial solution cardinality operates on conjunctions with variables. An example is the following expression identifying a row in Connect Four that could not be ground-instantiated:

```
1 true(cell(X, Y1, x)), succ(Y1, Y2),
2 true(cell(X, Y2, x)), succ(Y2, Y3),
3 true(cell(X, Y3, x)), succ(Y3, Y4),
4 true(cell(X, Y4, x)
```

Other than solution cardinality that indicates the stability of a true evaluation, partial solution cardinality indicates the negative stability of a false evaluation: The more partial solutions exist, the less stable is the expression evaluation to false.

In other words: For any conjunct ϵ of the form $(b_1 \wedge \ldots \wedge b_n)$ where there are at most d conjuncts fulfilled and there are b solutions (i.e., instantiations) with d conjuncts fulfilled we can state that the expression ϵ is more likely to hold in a future state for higher b and d:

$$P(holds(\epsilon, t)|t \in T(s) \wedge holds(\neg\epsilon, s)) \sim b$$
$$P(holds(\epsilon, t)|t \in T(s) \wedge holds(\neg\epsilon, s)) \sim d$$

Again, the feature is build on the assumption that only few expressions change their evaluation per state. The feature is comparable to the fuzzy evaluation of its ground-instantiated expression that would yield a disjunctions of conjunctions.

Note that the feature should be evaluated carefully as the order of instantiating each conjunct is crucial to run time: Given the above Connect Four example and assuming a 7x6 board, an evaluation from left to right yields only 7 possible instantiations for the variable Y_1 while all other variables Y_2, Y_3, Y_4 are defined in terms of Y_1. The worst possible instantiation would first instantiate the fluents and result in $7^4 = 2401$ possible instantiations for the Y_i. Given that humans often encode rules in a way that is efficient for evaluation, it is thus risky to deviate from left-to-right evaluation.

For this reason we chose the variable names b (breadth) and d (depth) which indicate the breadth and depth of solutions in the tree spanned when proving each single conjunct.

4.3.3. Distance Features

Distance features capture the pattern of evolution of a formula. We use the term evolution to express that, for instance, the advance of a pawn in Chess evolves or transforms a true formula $true(cell(c, 3, pawn))$ in a state s to another true formula $true(cell(c, 4, pawn))$ in its successor state s'. This implies that the former formula does not hold in the latter state and the latter formula did not hold in the former state.

As the base scenario, we will discuss the distance between atoms. All other distances are variants of this base scenario.

The distances calculated between atoms indicate lower bounds on the probability that an atom becomes true. The idea is that if an atom $true(cell(X, 8, pawn))$ is the goal, for example for queening a pawn, then a currently true atom $true(cell(X, 7, pawn))$ indicates a higher probability to lead to the goal than if the atom $true(cell(X, 1, pawn))$ would hold. In other words, the evaluation of $true(cell(X, 8, pawn))$ to false is much less stable given $true(cell(X, 7, pawn))$ than given $true(cell(X, 1, pawn))$.

We estimate these probabilities using distances. As distance measure we estimate the number of steps an atom a requires to evolve to another atom b. Since this number of steps is a lower bound on the number of steps required for transforming a to b, we can assume that the probability for reaching (a true evaluation of) b given (a true evaluation of) a is lower if the distance from a to b is higher.

As all GGP agents applying distance, we quietly (and not necessarily correctly) assume the opposite of this statement: A low lower bound on the number of steps indicates a high probability of b to hold in a future state.

Given two atoms a and b and a distance measure Δ between them, we can summarize that the probability of reaching atom b given atom a in the current state is indirectly proportional to the distance from a to b.

$$P(holds(b, t)|t \in T(s) \wedge holds(a, s)) \sim \frac{1}{\Delta(a, b)}$$

Note that $holds(\neg b, s)$ is an important precondition for the above formula to hold. We omitted it for the sake of readability. Since the confirmation of a distance feature is a non-trivial complex subject by itself, we describe how to confirm (and evaluate) distance features in section 4.3.5.

Variable Distance

The variable distance feature can be detected on conjunctions with variables. The following is an example of the game Pac Man where `inky` has the goal to catch `pacman`, expressed by both occupying the same square.

```
1 true(location(inky, X, Y)), true(location(pacman, X, Y))
```

Another variant is the following formula occurring in Othello:

```
1 count(1, 1, X, X)
```

Both cases are instances of the more general formula

$$\epsilon = \epsilon_r \wedge equal(X_1, X_2)$$

ϵ_r here serves for instantiating the variables X_1 and X_2 while $equal(X_1, X_2)$ unifies them.

Correspondingly, we motivate the variable distance feature by stating

$$P(holds(\epsilon_r \wedge equal(X_1, X_2), t) | t \in T(s) \wedge holds(\epsilon_r \wedge \neg equal(X_1, X_2), s)) \sim \frac{1}{\Delta(X_1, X_2)}$$

There are three problems that can arise when using variable distance features:

- Often, we do not know whether X_1 "moves into the direction of" X_2 and/or the other way round. Therefore, we calculate the distances $\Delta(X_1, X_2)$ and $\Delta(X_2, X_1)$ and use whatever value is lower.
- If the formula can be decomposed into more than one equality (as, e.g., in the above Pac Man example), we evaluate each single distance and aggregate it to an overall distance. The process will be described in detail in section 4.3.5.
- There may be various solutions to X_1 and/or X_2. A reasonable approach and our current solution is to use the minimum of all distances determined between instances of X_1 and X_2 since this represents the estimate for the minimum number of time steps needed to have $(X_1 = X_2)$ hold. We will discuss the aggregation of distances over various solutions again in section 4.4.1.

Note that distance only relates to the stability of an expression being false: The lower the distance between the variables, the less stable the false evaluation becomes. Consequently, the probability of evaluating to true in a future state rises.

Fixed Distance

The fixed distance feature is a specialization of the variable distance: Since one of the formulas ("the destination" part of the formula) is fixed, the source variables are instantiated and the destination part of the formula is omitted altogether.

Fixed distances features operate thus on formulas where the partially ordered symbols are ground, as, e.g., in Pac Man:

```
1 true(step(50))
```

The feature provides information on future states for the same reason as variable distance.

Order

Order features are used in Fluxplayer and also appear as relative features derived from other features in Cluneplayer. It compares two values with each other and confirms whether an order predicate holds on these. It can be detected in conjunctions with shared variables. An example is the following expression in `checkers`:

```
true(piece_count(white, W)),
true(piece_count(black, B)),
greater(W, B).
```

The feature expresses the idea that a comparison with a higher margin leads to a higher stability of the expression evaluation. The margin is relevant regardless of whether the expression holds or not.

Thus the feature relates to stability of both true and false evaluation. Given an order predicate *porder* and two arguments x, y we can state:

$$P(holds(porder(x,y),t)|t \in T(s) \wedge holds(porder(x,y),s)) \sim \Delta(x,y) \text{ and}$$
$$P(holds(porder(x,y),t)|t \in T(s) \wedge holds(\neg porder(x,y),s)) \sim \frac{1}{\Delta(y,x)}$$

4.3.4. Persistence Features

Persistence features do not involve a heuristic judgment of probabilities but (proved) facts. Once the fact holds, it is said to persist through the remainder of the match, hence the name. This is different from formulas that are always true or false since parts of the persistent expression depend on the current state. Therefore, it is not possible to evaluate the formula right away and assumptions on the state are required.

We discover persistence in basic fluent queries, that is, formulas in the form of an atom `true(f)` , however, other formulas may become persistent as a consequence.

In order to use persistence features, we need to prove that

True Persistence $true(f) \Rightarrow next(f)$

False Persistence $\neg true(f) \Rightarrow \neg next(f)$

for a given game description D and fluent f. For the proof we rely on [TV10, HT10] that inductively prove the property using a temporal extension of GDL.

For convenience we often call a fluent `f` true-persistent fluent and mean that an atom `true(f)`, once it evaluates to true in a state s, remains true in every (direct and indirect) successor state of s. Likewise, when we speak of a false-persistent fluent, we refer to a fluent `f` where the atom `true(f)`, once evaluated to false, remains false for the remainder of the match.

True Persistence

An example for a true-persistent fluent occurs in the game Tic-Tac-Toe.

```
true(cell(1,1,x))
```

Once a player marked a cell with an `x` the fluent persists for the remainder of the match, i.e., it holds in every future state state.

A feature is motivated by the fact that for any true-persistent fluent f it holds:

$$P(holds(true(f), t)|holds(true(f), s) \wedge t \in T(s)) = 1$$

and similarly

$$P(holds(\neg true(f), t)|holds(true(f), s) \wedge t \in T(s)) = 0$$

False Persistence

An example for a false-persistent fluent can also be found in Tic-Tac-Toe.

```
1 cell(1,1,b)
```

The rule states that a fluent remains unmarked (`b` = "blank") until it is marked. Once marked, it remains false through the remainder of the match.

Thus, for any false-persistent fluent f it holds:

$$P(holds(true(f), t)|t \in T(s) \wedge holds(\neg true(f), s)) = 0$$
$$P(holds(\neg true(f), t)|t \in T(s) \wedge holds(\neg true(f), s)) = 1$$

4.3.5. Confirming Distance Features

We confirm that a formula can be interpreted as a distance feature by determining a distance graph for the relevant atom in the formula. For introductory purposes we assume that there is exactly one relevant atom per formula. This relevant atom can be informally described as the "destination atom" representing the goal to be reached.

The distance graph for this atom is then a graph containing all possible "states" of the atom where an atom a is connected to an atom b if atom a may evolve to atom b in one time step.

So if the destination atom is $true(cell(a, 8, pawn))$ and we find that $true(cell(a, 7, pawn))$ can be transformed to the destination atom, then we add the corresponding edge to the graph.

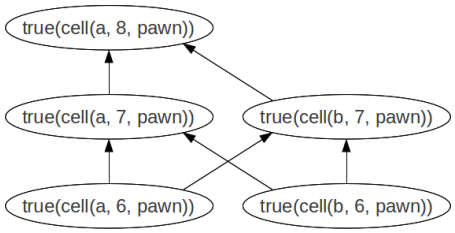


Figure 4.1.: Partial Distance Graph for $true(cell(a, 8, pawn))$

Figure 4.1 is a partial distance graph for a pawn in a Chess variant without promotion.

The distance between two atoms in the graph would be ideally represented by their temporal distance, that is, a distance based on the number of time steps required to

obtain the desired atom. While we present such an approach in section 4.5, for now we use the standard approach for determining distances as used by Cluneplayer, Fluxplayer and Kuhlplayer by relying on "symbol distances" [Clu07].

Symbol distances are estimated as the aggregation of the distances between the arguments of the atoms. Thus we can construct a distance graph for atoms if we can construct a distance graph for (at least one of) their arguments. The distance between two atoms $a = a(x_1, \ldots x_n)$ and $b = b(y_1, \ldots, y_n)$ can then be defined as

$$\Delta_{Atom}(a, b) \stackrel{\text{def}}{=} \bigotimes_i \Delta_{Arg}(x_i, y_i)$$

The function $\bigotimes$ aggregates the argument distances $\Delta_{Arg,i}$ to a single distance.

For now we set $\bigotimes = \sum$ to obtain the Manhattan distance. The resulting atom distance Δ_{Atom} can be used as a proxy for temporal distance that in turn is used for an estimation on how likely the expression is to become true (given that it is currently evaluated to false).

Argument Distances

In order to calculate $\Delta_{Arg,i}$ for a given argument index i, we look for a binary predicate $p_{Arg,i}$ that imposes a partial oder on the domain of the i^{th} argument. If we find such a predicate $p_{Arg,i}$, we construct a directed graph G_i where there is an edge from node n_1 to n_2 if $p_{Arg,i}(n_1, n_2)$ holds.

We then define the graph distance $\Delta_{Gi}(x_i, y_i)$ as the length of the shortest path from node x_i to node y_i in G_i or infinite (∞) if no such path exists.

The argument distance for the i^{th} argument can then be defined as

$$\Delta_{Arg}(x_i, y_i) \stackrel{\text{def}}{=} \begin{cases} 0 & x_i = y_i \\ \Delta_{Gi}(x_i, y_i) & \exists G_i \\ \infty & else \end{cases}$$

Thus, an argument distance $\Delta_{Arg,i}$ is based on graph distance Δ_{Gi} only if a corresponding graph G_i exists. Otherwise it defaults to 0 or ∞. Consequently, we only confirm a distance feature if there is at least one non-trivial argument distance, that is, if

- there is at least one argument position $1 \leq i \leq n$ where a graph G_i can be built and
- there is a shortest path between two nodes in G_i of at least length 1.

We can now define the different evaluations depending on the feature type and the formula ϵ from which the feature is derived. As part of this, we also define the predicate $p_{Arg,i}$ we use to construct G_i.

Variable Distances Between Fluents

For a formula ϵ of the form

```
true(f), true(g)
```

where

- $f(x_1, \ldots, x_n)$ and $g(y_1, \ldots, y_n)$ are fluents of common name and arity n and
- for some argument positions $i \in I_V$ in the non-empty set $I_V \subseteq \{1, \ldots, n\}$ each $x_i = y_i$ is a variable

the evaluation of the expression is defined as

$$\Delta(f, g) = \bigotimes_{i \in I_V} (\Delta_{Arg,i}(x_i, y_i))$$

where $\bigotimes = \sum$ is set to obtain the Manhattan distance.

The predicate $p_{Arg,i}$ for constructing G_i is obtained by finding all binary, static (i.e., not state-dependent) and antisymmetric predicates $pbinary \in \mathcal{B}$ in rules of the form

```
next(f(.., X2, ..)) :-
  true(f(.., X1, ..),
  ..,
  pbinary(X1, X2).
```

where the variables `X1, X2` occur at argument position i of fluent f.

We then define $p_{Arg,i}(x, y) \equiv \exists pbinary \in \mathcal{B} : pbinary(x, y)$

Note that binary predicates with switched argument positions should be included in $\mathcal{B}$ similarly.

Variable Distances in a Single Predicate

For a formula ϵ of the form

```
pdistance(.., X, .., X, ..)
```

where

- `X` is a variable that occurs at exactly two argument positions i_1 and i_2 and
- the predicate `pdistance` is recursive and state-dependent

we define the evaluation of the formula as $p_{Arg}(Y1, Y2)$ where $p_{Arg}(Y1, Y2)$ is the argument distance between `Y1` and `Y2` for the original expression ϵ but with `Y1` substituting `X` at position i_1 and `Y2` substituting `X` at i_2.

The predicate p_{Arg} for constructing G is obtained by finding all binary, static and antisymmetric predicates $pbinary \in \mathcal{B}$ in rules of the form

```
pdistance(.., X2, .., ) :-
  pbinary(X1, X2),
  ..,
  pdistance(.., X1, ..)
```

where the variables `X1, X2` occur at argument position i_1 or i_2 of predicate `pdistance`.

We then define $p_{Arg}(x, y) \equiv \exists pbinary \in \mathcal{B} : pbinary(x, y)$. As for distances between fluents, note that binary predicates with switched argument positions should be included in $\mathcal{B}$ similarly.

Fixed Distances

For a formula ϵ of the form

```
true(f)
```

the evaluation is the minimum distance between any fluent in the current state to f or ∞ if no suitable fluent is in the current state. The distance between fluents is obtained as for variable distances.

Order

For a formula ϵ of the form

```
.., porder(X, Y), ..
```

where

- `X, Y` are variables
- the predicate `porder` is static and binary
- the relation encoded by `porder(x,y)` is antisymmetric and transitive

we define p_{Arg} as the underlying partial order whose transitive closure coincides with `porder(x,y)`. In other words, we define $p_{Arg}(X, Y)$ as the subset of `porder(x,y)` for which there is no element Z that comes after X and before Y:

$$p_{Arg}(X, Y) \stackrel{\text{def}}{=} porder(X, Y) \text{ such that}$$
$$\nexists Z : (porder(X, Z) \wedge porder(Z, Y) \wedge (X \neq Z) \wedge (Z \neq Y))$$

The evaluation o of `porder` is then defined as

$$o = \begin{cases} -\Delta_{Arg}(x, y) & porder(X, Y) \\ \Delta_{Arg}(x, y) & porder(Y, X) \\ \infty & else \end{cases}$$

We also use o as evaluation for the complete formula ϵ.

4.3.6. Summary

	Scope	True Evaluation	False Evaluation
Solution Cardinality	Conjunctions	Correlates with Number of Solutions	-
Partial Solution Cardinality	Conjunctions with $n > 1$ Conjuncts	-	Correlates Negatively with Number/Depth of Partial Solutions
Order	Conjunctions	Correlates with Margin of Comparison	Correlates with Margin of Comparison
Variable Distance	Conjunctions	-	Correlates Negatively with Distance
Fixed Distance	Fluent Queries	-	Correlates Negatively with Distance
True Persistence	Ground Fluent Queries	Persists	-
False Persistence	Ground Fluent Queries	-	Persists

Table 4.2.: Rule-Derived Features: Detection

Table 4.2 shows for each feature the most specific type of candidate formula it works on and the relation of the feature to the stability of a true/false evaluation.

4.4. Integration of Rule-Derived Features

In the prior sections we covered how features should be interpreted and used and what type of features we detect. This section concludes the idea laid out by integrating the features in our fuzzy evaluation function.

As a consequence, this section will be tailored more towards our agent "Nexplayer" and also be more practical. Again, parts of this section were derived from our workshop contribution "Heuristic Interpretation of Predicate Logic Expressions in General Game Playing" to the GIGA Workshop 2011 [Mic11a].

In order to deal with feature integration, we will first discuss the feature output provided by the features discussed previously. This is necessary in order to normalize the feature output (section 4.4.1) and integrate it into a fuzzy evaluation function (section 4.4.2).

We conclude with an evaluation that indicates the generality and the effectiveness of our feature acquisition approach.

4.4.1. Feature Normalization

Feature normalization is required for two reasons. By normalizing features we predefine a contract or interface that facilitates integration in an aggregation function. In our case, we will normalize the output of each feature to a real value in the interval $[-1, 1]$.

And second, we provide a context for the output of each feature. For instance, if solution cardinality is applied on the pieces in chess, then a cardinality of 2 for pawns is quite low while a cardinality of 2 for queens is high. Obviously, the quantity of queens must be viewed differently than the quantity of pawns.

In order to define the output of the feature, we rely on the functions defined in the previous section when discussing the corresponding feature. Most often the functions appeared on the right hand side of the motivating probability expression. Normalization takes these functions and produces a normalized feature $\bar{\phi}(s)$ that takes a state s and returns a value in $[-1, 1]$. For convenience, we omit the state argument if doing so does not introduce ambiguities.

Solution Cardinality

The solution cardinality of an expression is the number of variable instantiations for ϵ such that $holds(\epsilon, s)$. In order to normalize this number, we could consider using the maximum number of solutions. However, it is difficult to find a low upper bound on solutions. A "syntactic" estimation of the maximum number of queens on a Chess board would yield 64 solutions, one for each square. A "semantic" estimation would yield 9 queens (8 promoted pawns + 1 queen). Still, conventional wisdom says that games with more than two queens are extremely rare.

Given this obvious mismatch between estimated and empiric upper bound, we decide to normalize the feature using the number of its solutions in the initial state s_0. We define the normalized feature in terms of the function $n(\epsilon, s)$ that returns the number of solutions to expression ϵ in state s.

$$\bar{\phi}(s) = 1 - 2^{\frac{1 - n(\epsilon, s)}{max(n(\epsilon, s_0), 1)}}$$

The exponential form guarantees strict monotonicity in the number of solutions.

Partial Solution Cardinality

Given a conjunction of n conjuncts, we evaluate the conjunction from left to right and determine how many conjuncts d we can fulfill and how many solutions (instantiations) b with d true conjuncts could be found.

For normalization, we use the maximum depth $d_{max} = n$ which is the number of conjuncts in the conjunction and the number of instantiations b_{max} that the conjunction can be fulfilled. Since b_{max} is hard to determine, we set it to a high constant ($b_{max} = 100$ in Nexplayer).

To facilitate things later, we interpret the feature output in the way of a normalized distance, meaning that values near 0 indicate expressions that are more likely to hold.

$$\bar{\phi} = 1 - \frac{d}{d_{max}} - \frac{b}{d_{max} * b_{max}}$$

Distance Features

Since each distance feature is defined in terms of argument distance, a first step towards normalizing distance features is normalizing the argument distance.

We recall that the argument distance was defined in terms of the shortest path in graph G from node x to node y.

$$\Delta_{Arg}(x,y) \stackrel{\text{def}}{=} \begin{cases} 0 & x = y \\ \Delta_G(x,y) & \exists G \\ \infty & else \end{cases}$$

In order to normalize the argument distance, we define $\bar{\Delta}_{Arg}(x,y)$ using the normalized graph distance $\Delta_G(x,y)$

$$\bar{\Delta}_{Arg}(x,y) \stackrel{\text{def}}{=} \begin{cases} 0 & x = y \\ \frac{\Delta_G(x,y)}{\max\limits_{z_1,z_2} \Delta_G(z_1,z_2)} & \exists G \\ \infty & else \end{cases}$$

such that the argument distance is always in $[0,1]$ or it is infinite (∞). Note that in the above term $\max\limits_{z_1,z_2} \Delta_G(z_1,z_2)$ we only assume finite maxima.

To obtain the normalized distance of one atom to another, we normalize the aggregation function (i.e., the Manhattan distance) by using the average instead of the sum of argument distances.

$$\bar{\Delta}_{Atom}(a(x_1,\ldots,x_n), b(y_1,\ldots,y_n)) \stackrel{\text{def}}{=} \frac{1}{n} \sum_i \bar{\Delta}_{Arg}(x_i, y_i)$$

Similarly, we set for all other occurrences $\bigotimes \stackrel{\text{def}}{=} \frac{1}{n}\sum_i^n$ if they aggregate normalized argument distances. If an argument distance $\bar{\Delta}_{Arg}$ is infinite, we evaluate $\bar{\Delta}_{Atom}$ as infinite. This follows the intuition that if an argument of an atom is unreachable, then the atom is unreachable.

Finally, to deal with more than one solution as they, e.g., could occur for variable distances and fixed distances, we need a minimum-like aggregation function that is strictly monotonic. The p-norm for $p < -1$ is such a function.

The overall distance $\bar{\Delta}_{overall}$ from any atom in the set X to any atom in the set Y is thus:

$$\bar{\Delta}_{overall} = \left(\sum_{x \in X, y \in Y} \bar{\Delta}_{Atom}(x,y)^p \right)^{\frac{1}{p}}$$

Infinite distances are implicitly taken care of if we assume that $\infty^p = 0$ for $p < 0$.

Note that the p-norm for negative p can be thought of as a set of parallel resistors where the overall resistance is sensitive to the quantity and resistance of each single resistor. Resistors of infinite resistance are ignored similarly. Still, if a new resistor is introduced into the circuit, the overall resistance decreases.

Persistence

The persistence features need not be normalized due to their boolean nature. We simply set the output of a true-persistent fluent to 1 and the output of a false-persistent fluent to -1.

4.4.2. From Feature Values to Fuzzy Values

The final step is the integration of the features into the fuzzy aggregation function.

For formula ϵ and a terminal state t we define four reference values

Stable Truth $\mathtt{t}$ is the value indicating that the expression will hold in the terminal state $P(holds(\epsilon, t)) = 1$

Probable Truth $\mathtt{t}_p$ is the value indicating that the expression will probably hold in the terminal state with $P(holds(\epsilon, t)) \geq 0.5$

Probable Falsity $\mathtt{f}_p$ is the value indicating that the expression probably hold in the terminal state with $P(holds(\epsilon, t)) \leq 0.5$

Stable Falsity $\mathtt{f}$ is the value indicating that the expression not hold in the terminal state $P(holds(\epsilon, t)) = 0$

To reflect the linear relationship we can assume

$$\mathtt{f} \leq \mathtt{f}_p \leq \mathtt{t}_p \leq \mathtt{t}$$

Our task is to define which feature maps to which interval of fuzzy values. To relate the values $\mathtt{f}_p$ and $\mathtt{t}_p$ to some human understandable concept, we assume that $\mathtt{t}_p$ corresponds to some *basic* probability that an expression holds in the terminal state and $\mathtt{f}_p$ corresponds to the probability of the complement. We assign to each interval of fuzzy values the following meaning

- $[\mathtt{t}_p, \mathtt{t}] : \epsilon$ is more likely to hold than the basic probability
- $[\mathtt{f}_p, \mathtt{t}_p] : \epsilon$ is undecided
- $[\mathtt{f}, \mathtt{f}_p] : \epsilon$ is more likely not to hold than the basic probability

In addition, we will assume a stability bias or inertia of states, that is

$$P(holds(\epsilon, t)|holds(\epsilon, s)) > P(holds(\epsilon, t)|holds(\neg\epsilon, s))$$

We then define the following mappings from normalized feature outputs to fuzzy values:

Solution Cardinality indicates at least probable truth, its values are thus mapped to $[\mathtt{t}_p, \mathtt{t})$.

Partial Solution Cardinality indicates more than probable falsity and less than probable truth, its values are thus mapped to $[\mathtt{f}_p, \mathtt{t}_p]$

Order indicates more than stable falsity and less than stable truth, we map it thus to $(\mathtt{f}, \mathtt{t})$

Distances indicates more than probable falsity and less than probable truth, its values are thus mapped to $[\mathtt{f}_p, \mathtt{t}_p]$

True Persistence indicates stable truth $\mathtt{t}$

False Persistence indicates stable falsity $\mathtt{f}$

Note that in some of the cases distances and partial solution cardinality it is difficult to determine " the right interval". Given that our features provide a heuristic on the underlying expressions, we decided to use the interval $[\mathtt{f}_p, \mathtt{t}_p]$ since it should be more probable to turn an expression true given a heuristic than given no heuristic.

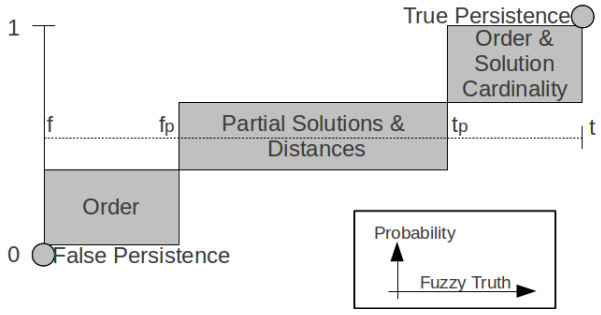


Figure 4.2.: Mapping Estimated Probabilities to Fuzzy Values

Figure 4.2 shows the mapping from probabilities that an expression holds in the terminal state and the corresponding fuzzy values we use. The features discussed realize parts of this mapping since they estimate the probabilities that we map to fuzzy values afterwards.

Table 4.3 shows for each feature and the formula evaluation it covers, what the corresponding feature values and fuzzy values are. The mapping is done linearly from the normalized feature values to the fuzzy values.

Since the output of a feature to its aggregation function must be defined for both cases of true and false expression evaluation, we evaluate the normalized feature only if the expression evaluation permits it. Otherwise, we return a default value.

So the normalized solution cardinality feature only produces an input to the aggregation function if the underlying expression holds in the current state. If it does not hold, then the default output of $\mathtt{f}_p$ is returned. Likewise, a normalized distance feature only

Feature	Formula Evaluation	Normalized Feature Value	Fuzzy Value
Solution Cardinality	True	$(0, 1)$	$(\mathtt{t}_p, \mathtt{t})$
Partial Solution Cardinality	False	$(0, 1]$	$[\mathtt{f}_p, \mathtt{t}_p)$
Order	False	$[-1, 0]$	$(\mathtt{f}, \mathtt{f}_p]$
	True	$[0, 1]$	$[\mathtt{t}_p, \mathtt{t})$
Variable Distance	False	$(0, 1]$	$[\mathtt{f}_p, \mathtt{t}_p)$
Fixed Distance	False	$(0, 1]$	$[\mathtt{f}_p, \mathtt{t}_p)$
True Persistence	True	1	$\mathtt{t}$
False Persistence	False	-1	$\mathtt{f}$
Default	False	-	$\mathtt{f}_p$
	True	-	$\mathtt{t}_p$

Table 4.3.: Range of Feature Output Values

produces input to the aggregation function if the underlying expression is evaluated to false. Otherwise, $\mathtt{t}_p$ must be returned.

So if m is a mapping from the normalized feature output $\bar{\phi}(s)$ to a fuzzy value, we define the feature output $\phi(s)$ for an expression ϵ as

$$\phi_\epsilon(s) = \begin{cases} m(\bar{\phi}^+(s)) & holds(\epsilon, s) \wedge \bar{\phi}^+ \text{ was detected on } \epsilon \\ m(\bar{\phi}^-(s)) & holds(\neg\epsilon, s) \wedge \bar{\phi}^- \text{ was detected on } \epsilon \\ \mathtt{t}_p & holds(\epsilon, s) \wedge \text{no feature } \bar{\phi}^+ \text{ was detected} \\ \mathtt{f}_p & holds(\neg\epsilon, s) \wedge \text{no feature } \bar{\phi}^- \text{ was detected} \end{cases}$$

The features $\bar{\phi}^+$ that correspond with an expression holding in a state are Solution Cardinality, Order and True Persistence. The features $\bar{\phi}^-$ that correspond with an expression not holding in a state are the distance features including Order and False Persistence.

In order to comply with our analysis of section 4.2, we will use no fixed values for $\mathtt{f}_p$ and $\mathtt{t}_p$ and instead adapt them to the degree of terminality of the state. This implies that we use a two-step state evaluation: First, we evaluate a state as to "how terminal" it is to determine $\mathtt{f}_p$ and $\mathtt{t}_p$. The closer the state is to fulfilling the terminal condition, the closer we set $\mathtt{f}_p$ and $\mathtt{t}_p$ to $\mathtt{f}$ and $\mathtt{t}$, respectively. Only then we evaluate the state and determine feature values.

4.4.3. Improvements

Before evaluating our approach, we discuss some improvements that increase quality, function efficiency and function construction efficiency for feature acquisition from rules.

Feature Combinations

Since most rule-derived features are only used if the underlying expression evaluates to a specific value, it is possible to combine features.

The output of combined feature is then defined as

$$\phi(s) = \begin{cases} \phi^+(\epsilon, s) & holds(\epsilon, s) \\ \phi^-(\epsilon, s) & holds(\neg\epsilon, s) \end{cases}$$

A prominent example is the evaluation of a conjunction. If the formula is false, we can apply partial solution cardinality, else solution cardinality.

In general, we can combine two features if they are applicable on the same expression and cover opposite evaluations of the underlying expression. For instance, partial solution cardinality can only be applied on conjunctions with variables, therefore it cannot be combined with a true persistence feature that operates on ground fluents. Moreover, although a Variable Distance feature also operates on conjunctions, an application together with a Partial Solution Cardinality feature would return a valid return value for both features and we would have to decide which one to use.

Among the possible combinations we have the following exceptions that do not make sense:

- False Persistence and True Persistence on a ground fluent do not make sense as the fluent in question either holds in the initial state and is therefore true-persistent or it does not hold and is therefore false-persistent.
- False Persistence and Fixed Distance features on ground fluents do not make sense as the former is implicitly expressed by the latter via an infinite distance.
- An Order feature cannot be combined with anything else since order heuristics determine the degree of truth and falsity of an expression.

In practice, we use

- Variable Distance features with Solution Cardinality conjunctions,
- Partial Solution Cardinality and Solution Cardinality on conjunctions, and
- Fixed Distance features with True Persistence on fluent queries

Of these three, the first is rather uncommon and we did not yet find a game where this was useful. The latter two occur frequently in games.

Lazy Ground Fluent Features

Since ground fluent queries occur only at the leafs of the aggregation function, the detection of features derived from ground fluents can be delayed. These are

- Fixed Distance Features and

- Persistence Features

Since both involve proofs, they are potentially costly. However, given that no other algorithm depends on them, we detect the features lazily, that is, we decide at run time how much start clock time is left and how we distribute time to obtain the features.

In practice, we found that distance features are detected faster, so we use all time for distance feature detection and assign the rest for persistence features.

Other Optimizations

Precalculated Distances Distances are calculated before the match starts and inserted into a lookup table to speed up evaluation.

Mutual Exclusion of True-Persistent Fluents If two (or more) true-persistent fluents cannot occur together in a state and one is evaluated to true, we can evaluate the others as if they were false-persistent. For instance, the fluents `cell(1,1,x)` and `cell(1,1,o)` are true-persistent and mutually exclusive.

Skipped Node Evaluation Once a false-persistent fluent holds in a state we evaluate, any conjunction with this fluent as positive (i.e., non-negated) conjunct also becomes false-persistent. Therefore, we can skip evaluating other conjuncts in the same conjunction and set the fuzzy output value for the represented rule to `f`. The same effect holds for negative occurrences of persistent true fluents. The optimization can be extended to disjunctions analogously.

Simplified Evaluation The same effect as in Skipped Node Evaluation can be compiled into the evaluation function once the current state (as opposed to a state encountered during search) includes false-persistent and true-persistent fluents.

The improvements of the evaluation function based on persistent fluents are also used for distance features that return an infinite distance, since this indicates that the fluent is persistent false. While we are aware that our detection mechanism for distance relationships is not correct, we did not yet find a game where the feature resulted in a mismatch.

4.4.4. Evaluation

We will evaluate the quality and efficiency of our feature acquisition mechanism in three major parts. In the first part, we will theoretically evaluate the criteria for evaluation functions on our feature acquisition mechanism to provide a background for our evaluation. We proceed by demonstrating the generality of our approach, detecting what features are applicable in which games. Finally, we will evaluate each group of features based on the results of matches of Nexplayer against Fluxplayer.

Analysis of Rule-Derived Feature Acquisition

In section 2.5 we defined key properties of evaluation functions. We discuss how these are fulfilled in each of the single features.

Efficiency of Feature Acquisition The efficiency of function construction is determined by the efficiency in feature acquisition. Our mechanism for feature detection returns only a small set of candidate features based on expressions. Most of these features require additional evidence such as underlying distance relationships or proofs of persistence that also represents the most time-consuming part of feature detection. This disadvantage is mitigated by the fact that evidence for distance and persistence features is typically common to a set of formulas due to abstract, i.e., non-ground, rules. This means that by acquiring, e.g., a distance graph for a specific ground fluent we obtain a graph that can be reused for other fluents of the same name and arity. We further reduce the computation costs incurred by allowing for lazy feature construction deciding at run time and based on the time remaining whether a feature should be analyzed.

All other operations commonly used by other agents, such as expensive simulations, are avoided: Selection can be skipped because we assume rule-derived features generally useful. Integration is not necessary due to the context provided by the aggregation function that embeds the underlying expression.

Efficiency of Feature Evaluation The efficiency of function evaluation is determined by the run time and number of our features. While persistent fluents are evaluated just like standard fluents, distance and solution cardinality features incur additional costs. We compare the cost of evaluating a feature to the cost of evaluating the underlying expression with a reasoner.

The cost for a solution cardinality feature is equal to the cost of finding multiple solutions to an expression. We can easily scale the evaluation cost by stopping to look for further solutions after a specific threshold is reached. Solution cardinality also represents a speed-up since it evaluates an expression that would have otherwise have been ground-instantiated. By employing solution cardinality, we thus reduce the overhead induced by the evaluation function.

Partial solution cardinality incurs equal costs as a standard expression evaluation plus an additional overhead per atom evaluation for keeping track of depth and number of solutions. The costs are thus roughly a multiple of standard expression evaluation by the reasoner.

Among all distance features the variable distance feature is most expensive since it represents $m * n$ distance calculations from m sources to n targets. For high m and n this could be a problem, however, we did not yet encounter such a situation.

Quality of Feature Evaluation Finally, the quality of state evaluation depends on the quality of feature evaluation and their integration. Each single feature is deterministic and thus consistent in its output. Moreover, we put great care to design features such that each minor change in a state, even though affecting the expression only indirectly, is

reflected in the feature output into the "right direction". An example is the application of p-norms for aggregating various distances instead of the minimum. In general, partial solutions, small distances or true persistence each result in an output closer to `t` if the underlying expression is supposed to hold. While this is not sufficient for fulfilling strict monotonicity, it also does not disprove it.

Finally, we evaluate each feature depending on the "degree of terminality" of the underlying state. For terminal states all features produce the values `f` or `t` and thus extrapolate terminal states.

Affected Games

We show that our feature acquisition method is general. For this purpose we demonstrate that the feature types are applicable in almost all games and that in each game many expressions are transformed to features.

We applied our feature acquisition mechanism on a set of 191 games largely equivalent to the games available on the GGP server [Sch12].

Figure 4.3.: Bitmap of Features vs. Games. The x-axis depicts 191 games in alphabetic order and the y-axis depicts the features in the following order (top-down): Solution Cardinality, Partial Solution Cardinality, Order, Variable Distances, Fixed Distances, True Persistence, False Persistence

Figure 4.3 shows a bitmap of features types versus games. On the x-axis all 191 games are depicted in alphabetic order, on the y-axis we see the seven feature types in the following order (top-down): Solution Cardinality, Partial Solution Cardinality, Order, Variable Distances, Fixed Distances, True Persistence, False Persistence. If at least one feature of the given type was found in a game, the pixel is marked black, otherwise white.

Figure 4.4 shows the same data as in the previous figure, aggregated by feature types per game. On the x-axis we see the number of different feature types detected per game. The y-axis indicates the corresponding number of games. We see that most games allow for the detection of two different types of features.

Table 4.4 shows the individual feature statistics. The first two columns show the number of games in which at least one feature of the given type was found and the corresponding relative frequency. The other two columns show, how often the feature was found in all games and the corresponding frequency of each feature per game. We can see that in average 2.3 different feature types are found per game and 25.4 features occur in average in the evaluation function of a game.

In summary, we can state that the proposed feature detection mechanism is very general and is able to detect different feature types in most games.

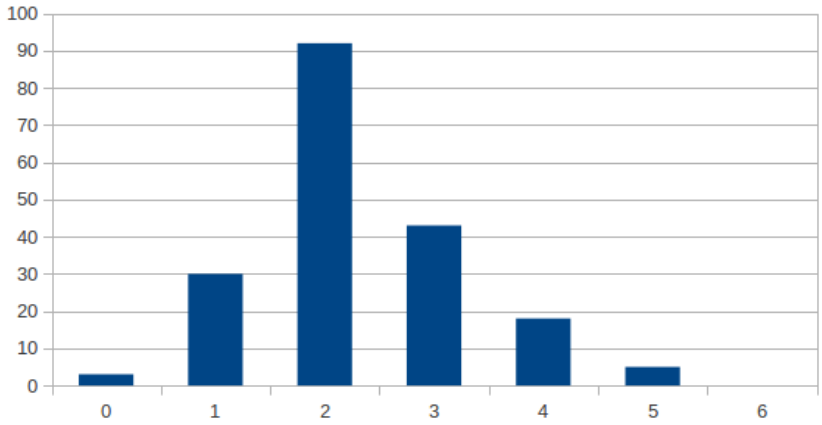


Figure 4.4.: Distribution of Games with Different Feature Types. Read: There are Y games in which exactly X different feature types are found.

Qualitative Evaluation - Solution Cardinality

To evaluate the solution cardinality features, we equip Nexplayer v1.1 with the corresponding acquisition mechanism. We refer to this version as Nexplayer v1.1s. We set up Nexplayer against Fluxplayer in a series of 20 matches in the games played in section 3.3.3. Time control was set to 40 seconds start clock and 4 seconds play clock.

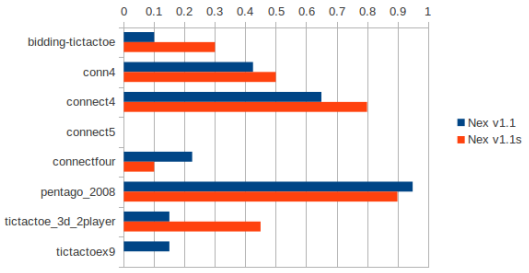


Figure 4.5.: Solution Cardinality - Win Rate of Nexplayer v1.1s and v1.1 against Fluxplayer

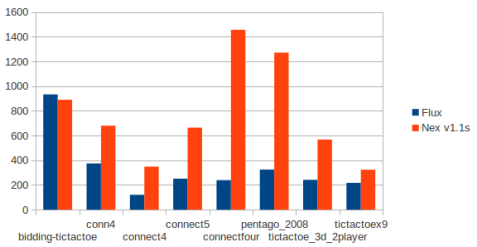


Figure 4.6.: Solution Cardinality - States per Second

Figures 4.5 and 4.6 shows the win rate achieved and the average number of unique states inspected per second. To provide for comparison of the win rate, we included the values achieved by Nexplayer v1.1 as obtained in section 3.3.3.

Nexplayer v1.1s achieves slightly better results than Nexplayer v1.1, averaging in total 38.1 points. The reason is that in Nexplayer v1.1 we evaluate expressions only as to whether they hold or not while Nexplayer v1.1s additionally counts the solutions and

	Found in Games	Frequency of Finding Feature	Number Features Found	Features Per Game
Solution Cardinality	85	44.5%	467	2.44
Partial Solution Cardinality	38	19.9%	345	1.81
Order	32	16.8%	106	0.55
Variable Distance	30	15.7%	43	0.23
Fixed Distance	114	59.7%	1056	5.53
True Persistence	118	61.8%	2641	13.83
False Persistence	23	12.0%	194	1.02
Total	440	230%	4852	25.4

Table 4.4.: Feature Detection Statistics

thus provides more accurate results.

Since both evaluations are almost identical and the number of solution cardinality features in these games small, speed is rarely affected.

Qualitative Evaluation - Persistence

To evaluate the persistence features, we equip Nexplayer v1.1 with the corresponding acquisition mechanism. We refer to this version as Nexplayer v1.1p. We set up Nexplayer against Fluxplayer in a series of 20 matches in the games played in section 3.3.3.

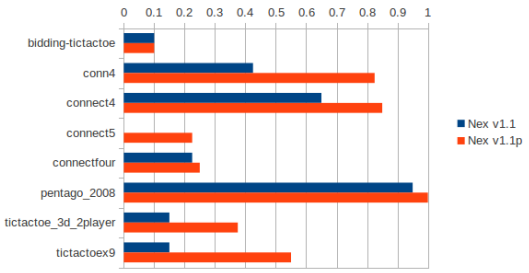


Figure 4.7.: Persistence - Win Rate of Nexplayer v1.1p and v1.1 against Fluxplayer

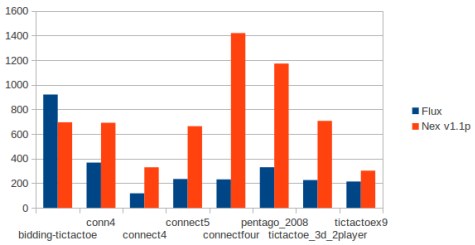


Figure 4.8.: Persistence - States per Second

Figures 4.7 and 4.8 show the average points achieved and the average number of unique states inspected per second. Again we included the values achieved by Nexplayer v1.1 as obtained in section 3.3.3 for comparison of the win rate.

Nexplayer v1.1p achieves much better results than Nexplayer v1.1 and achieves approximate parity against Fluxplayer. Evaluation speed is in some cases minimally slower which we attribute to the fixed cost of proving persistence (typically between 1 and 10

seconds). In other cases it is significantly higher due to the removal of persistent false conjunctions and persistent true disjunctions from the evaluation function.

Qualitative Evaluation - Distances

For the evaluation of distance features, we equip Nexplayer v1.1 with the corresponding acquisition mechanism. We refer to this version as Nexplayer v1.1d. We set up Nexplayer against Fluxplayer in a series of 20 matches in a set of distance-related games.

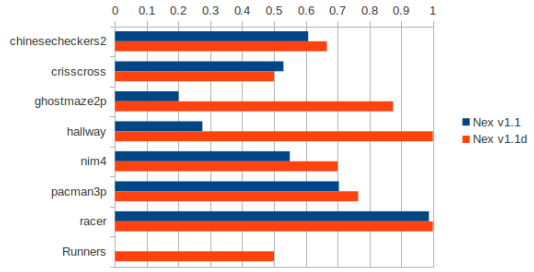


Figure 4.9.: Distance - Win Rate of Nexplayer v1.1d and v1.1 against Fluxplayer

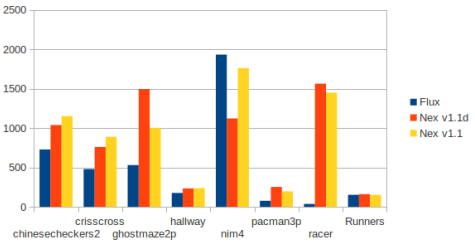


Figure 4.10.: Distance - States per Second

Figures 4.9 and 4.10 show the win rate of Nexplayer v1.1d in the distance-related games and the average number of unique state encountered per second. For comparison, we ran the same evaluation for Nexplayer v1.1 without distance features and included it in the above figures. Note that given games are not constant-sum games and points per match do not necessarily add up to 100.

We see that Nexplayer v1.1d performs significantly better than both Nexplayer v1.1 and Fluxplayer. At the same time, evaluation efficiency is reasonable and does not significantly deviate from that of the other agents.

Summary

Table 4.5 summarizes the different evaluations.

	Nexplayer Version			
	v1.1	v1.1s	v1.1d	v1.1p
Structural Games	33.12	38.13	-	52.19
Distance Games	49.73	-	80.14	-

Table 4.5.: Average Points of Nexplayer Versions against Fluxplayer

We can see in each case that a small start clock of 40 seconds is sufficient to construct our features. We did not reduce the start clock to the possible limit since our primary intention was to evaluate quality. Given that each single feature improves the playing

strength of Nexplayer while imposing minor time constraints on construction and evaluation, we expect an agent combining the features to be competitive against other agents. We will address both open points (i.e., reduced start clock and combined features) as part of the general evaluation and refer the impatient reader to chapter 5.

Although more general than other distance detection mechanisms, one disadvantage of our feature acquisition is the dependence of distance features on syntactic circumstances. For instance, the predicate p_{Arg} used for constructing a graph G in order to obtain Δ_G was defined in terms of binary predicates in `next` rules and rules of recursive predicates. Rules such as

```
1 next(f(Y)) :- true(f(X)), plus(X, 1, Y)
```

can thus not be detected, because `plus` has arity 3. The following are other examples that do not work with our approach:

- distances across different fluent symbols, e.g., `pawn(8,7)` to `queen(8,8)`
- distances across different arguments, e.g., when moving object b three elements to the right: `row(b,a,a,a,a)` to `row(a,a,a,b,a)`
- distances based on content, e.g., `cell(5,5,fixed)` to `cell(5,6,fixed)` which should not be the same as `cell(5,5,movable)` to `cell(5,6,movable)` for a fixed board element `fixed` and a movable board element `movable`.

We will address these problems in the next section.

4.5. Acquisition of Admissible Distance Features

In the prior section we discussed rule-derived features and, as part of these features, distance features.

We found that distances can be used to evaluate the "reachability" of formulas based on information about its pattern of evolution. Distance is then an estimate for the lower bound on the number of steps to reach a state in which the desired formula holds. Given that matches terminate after a finite number of steps, higher distances indicate lower probabilities that the formula eventually holds. For our heuristic we adopted the general assumption that lower distances indicate a higher probability that the formula becomes true.

While we stated that the ideal distance measure would be the number of time steps towards the formula, we used distances based on symbols. Such symbol distances are used by all deterministic GGP agents and determine the distance between, e.g., two fluents as the aggregated distance between the symbols of each fluent argument.

Though being a suitable distance heuristic with significant impact on the playing strength, the approach still has disadvantages.

- The detection of distances depends on syntactic patterns. Specifically binary static predicates play an important role. If the information is contained in a non-binary predicate or contained in a ground-instantiated form, the distance feature detection fails.

- Distances between two predicates or fluents are calculated as (normalized) Manhattan distance over the distances between each argument. Consequently, distance values obtained do not depend on the move pattern of the 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 addition to being imprecise, this also renders the heuristic inadmissible.

In this section we will present an algorithm that allows for the calculation of distances without the aforementioned problems. The underlying idea is a comprehensive analysis of the `next` rules of game in order to find dependencies between entire fluents. Thus, while being specifically tailored towards ground-instantiated fluents (and not predicates), it allows to avoid the detour of using fluent arguments. 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. In contrast to previous approaches, our approach does not depend on syntactic patterns.

The remainder of this section is structured as follows: In section 4.5.1 we introduce fluent graphs as the theoretical basis for this work and show how to use them to derive distances from states to fluents. We proceed in section 4.5.2 by showing how fluent graphs can be constructed from a game description and discuss their evaluation and normalization in section 4.5.3. Finally, we conduct experiments in section 4.5.4 to show the benefit and generality of our approach.

The remainder of this section is derived from our workshop contribution "Distance Features in General Game Playing" to the GIGA Workshop 2011 [MS11] and its follow-up conference article with the same name at ICAART 2012 [MS12]. Both submissions were joint work with my former colleague and fellow PhD student Stephan Schiffel.

4.5.1. 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 4.11 shows a partial fluent graph for Tic-Tac-Toe that relates the fluents `cell(3,1,Z)` for `Z` ∈ `{b, x, o}`.

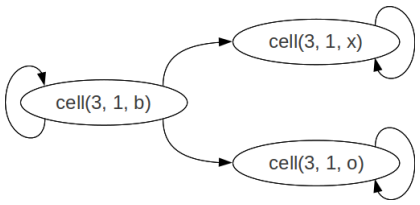


Figure 4.11.: Partial Fluent Graph for Tic-Tac-Toe

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 fluent `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 4.1 (Fluent Graph)**.** Let Γ be a game over a game description D with the 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'

$$\begin{aligned}&(s' \text{ is a successor of } s) \wedge f' \in s' \\ &\quad \Rightarrow (\exists f)(f, f') \in E \wedge (f \in s \cup \{\emptyset\})\end{aligned} \tag{4.5.1}$$

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 4.12 shows. We will address this problem later.

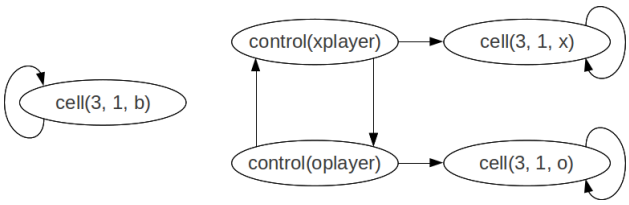


Figure 4.12.: Alternative partial fluent graph for Tic-Tac-Toe

We can now define a distance function between two fluents as follows:

Definition 4.2 (Fluent 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

$$\Delta_F(f, f') \stackrel{\text{def}}{=} \begin{cases} 0 & f = f' \\ \Delta_G(f, f') & else \end{cases}$$

In order to use the distance, e.g., to determine the distance from the current state to a state in which fluent f' holds we define additionally a distance function $\Delta_{s \to F}(s, f')$ in terms of the fluent distance. For aggregating various fluent distances, we choose the minimum function.

Definition 4.3 (State Distance Function)**.** Let $\Delta_F(f, f')$ be a fluent distance function. Then

$$\Delta_{s \to F}(s, f') = \min_{f \in s \cup \{\emptyset\}} \Delta_F(f, f')$$

That means, we compute the distance $\Delta_{s \to F}(s, f')$ as the shortest path in the fluent graph from any fluent in s (or $\emptyset$) 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 $\Delta_{s \to F}(s, f')$ 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 $\Delta_{s \to F}(s, f')$ is an admissible heuristic for reaching f' from a state s.

Theorem 4 (Admissible Distance)**.** Let

- $\Gamma = (R, s_0, T, l, u, g)$ be a game with ground terms Σ and states $\mathcal{S}$,
- $s_0 \in \mathcal{S}$ 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:

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

If $\Delta_{s \to F}(s_1, f) = n$, 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 $\Delta_{s \to F}(s_1, f) = n$ 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 4.1, 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, $\Delta_{s \to F}(s_1, f) \leq m$ because $f_j \in s_1 \cup \{\emptyset\}$ and $f_{m+1} = f$. However, $\Delta_{s \to F}(s_1, f) \leq m$ together with $m < n$ contradicts $\Delta_{s \to F}(s_1, f) = n$. □

4.5.2. 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 $\mathcal{N}$ of the formulas that imply $\texttt{next}(f')$.
2. For every disjunct $\mathcal{N}_\mathcal{C}$ in $\mathcal{N}$:
   - Pick one literal $\texttt{true}(f)$ from $\mathcal{N}_\mathcal{C}$ 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 $\mathcal{N}$. Since, the distance function $\Delta_{s \to F}(s, f')$ 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 $\mathcal{N}$ 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.

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

A formula $\mathcal{N}$ in DNF is a set of formulas $\{\mathcal{N}_{\mathcal{C}1}, \ldots, \mathcal{N}_{\mathcal{C}n}\}$ connected by disjunctions such that each formula $\mathcal{N}_{\mathcal{C}i}$ is a set of literals connected by conjunctions. We propose the algorithm in figure 1 to construct $\mathcal{N}$ such that $\texttt{next}(f') \Rightarrow \mathcal{N}$.

Algorithm 1 Constructing a formula $\mathcal{N}$ in DNF with $\texttt{next}(f') \Rightarrow \mathcal{N}$.

Input: game description D, ground fluent f'
Output: $\mathcal{N}$, such that $\texttt{next}(f') \Rightarrow \mathcal{N}$

1: $\mathcal{N} := \texttt{next}(f')$
2: $finished := false$
3: **while** $\neg finished$ **do**
4: Replace every positive occurrence of $\texttt{does}(r, a)$ in $\mathcal{N}$ with $\texttt{legal}(r, a)$.
5: Select a positive literal l from $\mathcal{N}$ 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: $\mathcal{N} := \mathcal{N}\{l/\hat{l}\}$
11: **end if**
12: **end while**
13: Transform $\mathcal{N}$ into disjunctive normal form, i.e., $\mathcal{N} = \mathcal{N}_{\mathcal{C}1} \vee \ldots \vee \mathcal{N}_{\mathcal{C}n}$ and each formula $\mathcal{N}_{\mathcal{C}i}$ is a conjunction of literals.
14: **for all** $\mathcal{N}_{\mathcal{C}i}$ in $\mathcal{N}$ **do**
15: Replace $\mathcal{N}_{\mathcal{C}i}$ in $\mathcal{N}$ by a disjunction of all ground instances of $\mathcal{N}_{\mathcal{C}i}$.
16: **end for**

The algorithm starts with $\mathcal{N} = \texttt{next}(f')$. Then, it selects a positive literal l in $\mathcal{N}$ and unrolls this literal, that is, it replaces l with the bodies of all rules $h\texttt{:-}b \in D$ 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 $\mathcal{N}$ into disjunctive normal form and replace each disjunct $\mathcal{N}_{\mathcal{C}_i}$ of $\mathcal{N}$ by a disjunction of all of its ground instances in order to get a ground formula $\mathcal{N}$.

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 $\mathcal{N}$. As a consequence the resulting formula $\mathcal{N}$ is not equivalent to $\texttt{next}(f')$. However, $\texttt{next}(f') \Rightarrow \mathcal{N}$, under the assumption that only legal moves can be executed, i.e., $\texttt{does}(r, a) \Rightarrow \texttt{legal}(r, a)$. This is sufficient for constructing a fluent graph from $\mathcal{N}$.

Note, that we do not select negative literals for unrolling. The algorithm could be easily adapted to also unroll negative literals. However, in the games we encountered so far, doing so does not improve the obtained fluent graphs but complicates the algorithm and increases the size of the created $\mathcal{N}$. Unrolling negative literals will mainly add negative preconditions to $\mathcal{N}$. However, negative preconditions are not used for the fluent graph because a fluent graph only contains positive preconditions of fluents as edges, according to Definition 4.1.

Selecting Preconditions for the Fluent Graph

If there are several literals of the form $\texttt{true}(f)$ in a disjunct $\mathcal{N}_{\mathcal{C}}$ of the formula $\mathcal{N}$ constructed above, we have to select one of them as source of the edge in the fluent graph. As already mentioned, the distance $\Delta_{s \to F}(s, f)$ 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 that is 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 difficult because it 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 $\mathcal{N}_{\mathcal{C}}$ 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':

$$sim(t,t') = \begin{cases} 1 & t, t' \text{ have arity 0 and } t = t' \\ \sum_i sim(t_i, t'_i) & t = f(t_1, \ldots, t_n) \text{ and} \\ & t' = f(t'_1, \ldots, t'_n) \\ 0 & else \end{cases}$$

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(2,3,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 $\mathcal{N}$. Thus we maximize the chance of reusing edges and minimize the number of edges in the graph.

4.5.3. Application in Nexplayer

For using the distance function in our evaluation function, we define the normalized distance $\bar{\Delta}_{s \to F}(s, f)$ in terms of the normalized fluent distance:

$$\bar{\Delta}_F(f, f') \stackrel{\text{def}}{=} \begin{cases} 0 & f = f' \\ \dfrac{\Delta_G(f, f')}{\max\limits_g \Delta_G(g, f')} & else \end{cases}$$

$$\bar{\Delta}_{s \to F}(s, f) = \bigotimes_{g \in s \cup \{\emptyset\}} \bar{\Delta}_F(g, f)$$

The value $\max_g \Delta_G(g, f)$ is the maximum length of any finite shortest path in G from any fluent g to f. Thus, the normalized distance $\bar{\Delta}_F$ will be in the interval $[0, 1]$ or infinite.

We define a feature upon the normalized distance similar to the distance of a standard feature, that is

- we use a p-norm with $p < -1$ to aggregate several distances to a single one (see section 4.4.1)
- we map the distance into the fuzzy range $[\mathtt{f}_p, \mathtt{t}_p)$ (as defined in section 4.4.2)

In comparison to the distance features presented in section 4.3, the admissible distance features are defined currently only on ground fluents. Therefore, only the Fixed Distance feature is implemented using admissible distances. However, admissible distances can be found on rules over implicitly defined domains, i.e., rules whose domain is a set of ground-instances "hard-coded" into the game description.

An example is the game Tic-Tac-Toe:

```
1 next(cell(X, Y, x)) :-
2   does(xplayer, mark(X, Y)), true(cell(X, Y, b)).
3 next(cell(X, Y, o)) :-
```

```
4   does(oplayer, mark(X, Y)), true(cell(X, Y, b)).
5 next(cell(X, Y, C)) :-
6   true(cell(X, Y, C)), distinct(C, b).
7 next(cell(X, Y, b)) :-
8   does(P, mark(M, N)), true(cell(X, Y, b)),
9   (distinct(M, X); distinct(N, Y)).
```

The above four rules encode the relation also illustrated in the fluent graph in figure 4.11, namely

$$(b, b), (b, x), (b, o), (o, o), (x, x)$$

The relation forms a partial order and thus allows for the application of distances. Notice how the corresponding distance graph indicates persistence information, that is, the fact that fluents with cell content b are false-persistent and fluents with cell content x or o are true-persistent.

4.5.4. Evaluation

We evaluate our algorithm for the two dimensions of efficiency and quality.

We begin by evaluating the time needed for feature acquisition which may increase substantially given the potentially large size of the fluent graph. We then analyze the effects of applying the admissible distance heuristics on the playing strength of an agent. For this purpose, we will compare the distance feature acquisition as defined in the previous section 4.3.3 and the admissible distance feature acquisition presented in this section. To avoid ambiguities, we refer to the former as non-admissible distance features and the latter as admissible distance features.

Efficiency of Feature Acquisition

We evaluate the time needed for feature acquisition on the complete set of games for all fluents that occur in the goal or terminal condition.

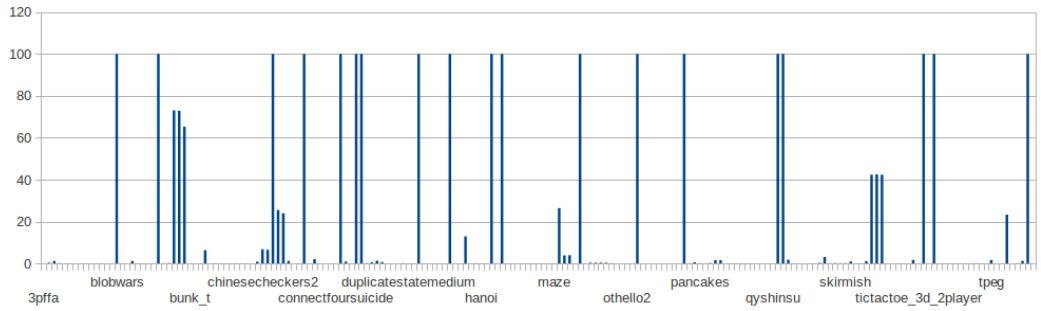


Figure 4.13.: Time for Detecting Admissible Distances

Figure 4.13 shows the time we need to find admissible distances. We capped the maximum time for detecting the features to 100 seconds. This should cover most competition

scenarios.

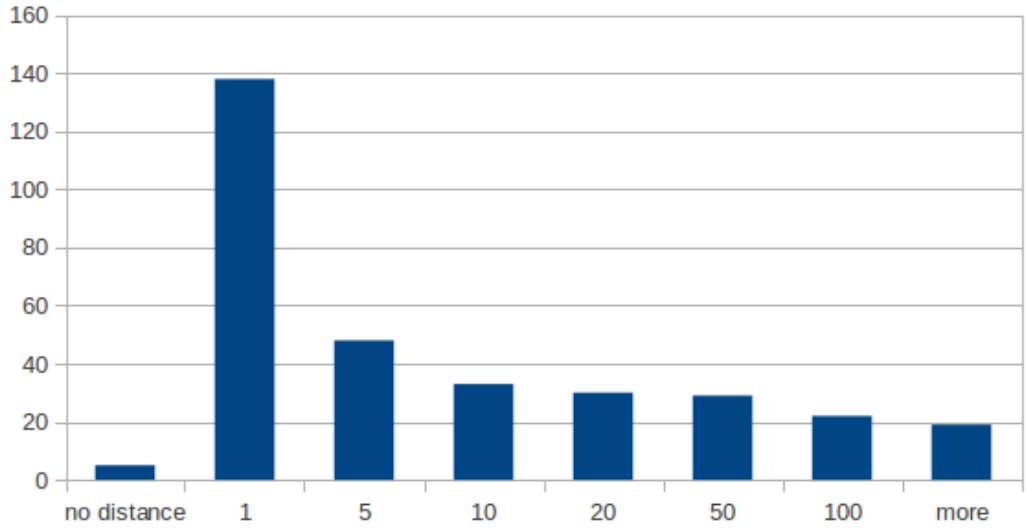


Figure 4.14.: Number of games grouped by the time required to find the admissible distance features. Read: There were Y games with admissible distance features detected in at most X seconds.

Figure 4.14 shows the same data but organized by the time required to construct the admissible distance features.

While most distance features can be detected quickly, there are 19 games (approx. 10 % of all games) that require more than 100 seconds.

Quality and Efficiency of Feature Evaluation

For evaluation, we define two variants of on top of our basic agent Nexplayer v1.1 as presented in section 3.3.

Nexplayer v1.1n is Nexplayer v1.1 that detects only the distance feature "Fixed Distance" (see section 4.3.3) and only on ground fluent queries. All other distance features are deactivated. In terms of features, Nexplayer v1.1n uses thus a subset of Nexplayer v1.1d.

Nexplayer v1.1a is Nexplayer v1.1 with all features deactivated except the detection of admissible distance features on ground fluent queries as presented in this section.

Both agents thus only detect distances over ground fluents and only differ in the approach on how to detect them.

We evaluated both agents in 20 matches with 120 seconds start clock and 4 seconds play clock against Fluxplayer. The start clock is much higher than usual to reflect the increased time requirements of the admissible feature detection.

As games we selected the 16 games used also for evaluating the previous Nexplayer version 1.1s, v1.1d and v1.1p (see section 4.4.4).

We split the set of games into two groups. The group of structural games includes mainly Tic-Tac-Toe and Connect Four variants and the group of distance games are the same games as used for evaluating Nexplayer v1.1d. The reason is that in structural games we expect Nexplayer v1.1a to perform better than Nexplayer v1.1n because it is able to detect distances over implicitly defined domains. On the other hand, in other distance-related games it is for the moment unclear to what extent the admissibility and high time requirements for construction influence the playing strength.

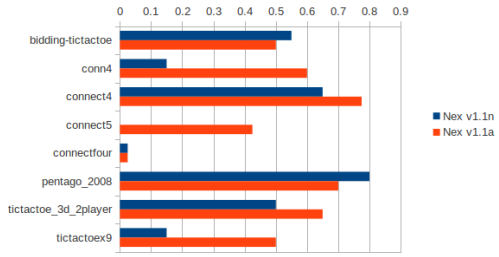


Figure 4.15.: Win Rates of Nexplayer v1.1a and v1.1n against Fluxplayer in Structural Games

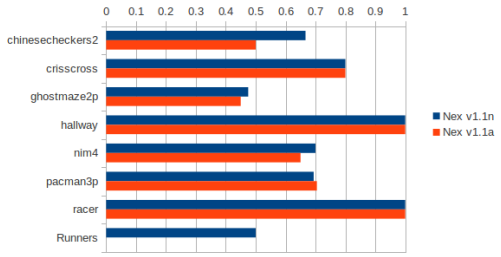


Figure 4.16.: Win Rates of Nexplayer v1.1a and v1.1n against Fluxplayer in Distance Games

Figures 4.15 and 4.16 show the win rates of both Nexplayer versions against Fluxplayer. While in the group of structural games v1.1a performs much better than v1.1n, it seems that the performance in the distance games is equal at best. Note that the game `Runners` here is a special case where negative distances (discussed in the appendix A.2.2) come into play. These are currently not detected for admissible distance features, rendering the heuristic bad. The corresponding value should correspondingly be ignored.

We can see the same pattern when setting up Nexplayer v1.1a directly against Nexplayer v1.1n. Figures 4.17 and 4.18 show the win rate of Nexplayer v1.1a against its non-admissible counterpart. Note here that the games `connectfour` and `pentago_2008` do not use admissible distances (all other structural games do), thus we interpret the minor differences in these games as random influences. For the structural games, `Runners` should be ignored for the reasons mentioned before. Also, although as it seems the relative performance of Naxplayer v1.1a in the game `racer` is bad, in fact, both agents do not gain any point and the value in the chart should correspondingly read “undefined”.

Taking these special cases into account, we see again that the admissible distance features increases the playing strength of the agent for structural games and slightly decrease it for distance games.

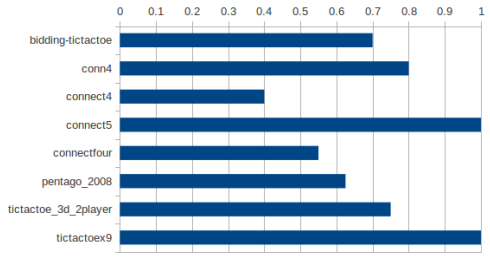


Figure 4.17.: Win Rates of Nexplayer v1.1a against Nexplayer v1.1n in Structural Games

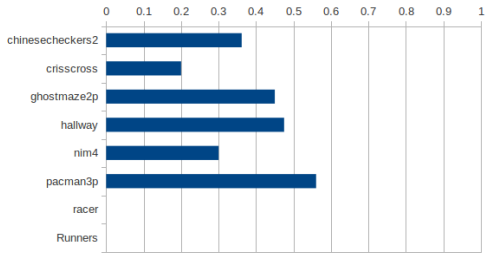


Figure 4.18.: Win Rates of Nexplayer v1.1a against Nexplayer v1.1n in Distance Games

	Structural Games			
	Flux	Nex v1.1n	Nex v1.1a	Avg
Flux	-	64.69	47.81	56.25
Nex v1.1n	35.31	-	27.19	31.25
Nex v1.1a	52.19	72.81	-	62.50
	Distance Games			
Flux	-	35.66	36.84	36.25
Nex v1.1n	75.06	-	63.04	69.05
Nex v1.1a	56.38	32.01	-	44.20

Table 4.6.: Average Points in Structural Games. The agent to the left won X points against the agent on the top.

Table 4.6 shows the average results of each agent for the two classes of games.

Figure 4.19 shows the average number of unique states analyzed per second in the matches against Fluxplayer. When looking at the numbers, we see that there is practically little difference in the evaluation time required by the admissible and non-admissible distance feature approaches.

Summary

Obviously, the acquisition of admissible distances is in some cases much more expensive than the previously introduced (non-admissible) distance feature detection. At the same time it is more general since it also detects distances over domains implicitly defined in the game rules, and it is more specific since it is currently only applicable for features over ground fluent queries.

Specifically, in the case for implicitly defined domains it increases the playing strength. On the other hand, if a non-admissible distance is found, Nexplayer's strength does not improve or even decrease using admissible distances. This shows that while Nexplayer indeed takes advantage of the provided distance in structural games, it does not benefit in any way from admissible features when compared to non-admissible features.

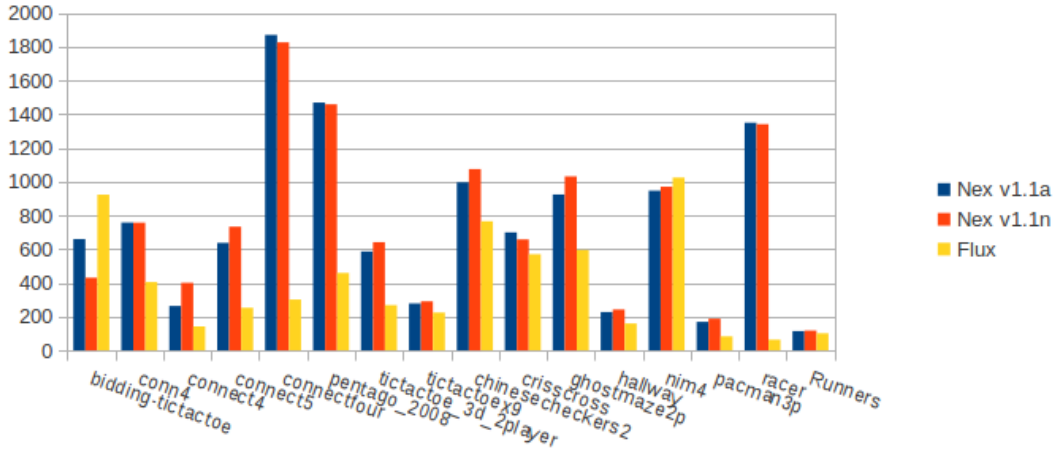


Figure 4.19.: Comparison of Unique States Analyzed per Second: Nexplayer v1.1a, v1.1n and Fluxplayer

In the light of these findings, it seems reasonable to apply admissible distances as a fallback mechanism that is used if no non-admissible distance is found. In this way, we benefit either way:

- If a non-admissible distance is found, we can skip detecting admissible distances since we (currently) do not benefit from the admissibility of the results.
- If no non-admissible distance is found, we invest the remaining time of the start clock to look for admissible distance features. If found, it is likely that they increase the quality of the evaluation.

Using this strategy, we avoid the problem of possibly high time requirements for admissible distance feature detection and restrict their application to the cases where we benefit most.

4.6. Summary

We have presented a categorization of feature acquisition methods along the lines of quality and efficiency. We found the best trade-off in generality and construction time to be rule-derived features.

Before applying them, we investigated the meaning of rule-derived features and found that they represent the evaluation of the underlying expression in the estimated terminal state. As an interesting side note we found that probabilistic Monte-Carlo evaluation is the sampling of a rule-derived feature with the goal formula as underlying expression.

We applied this view by motivating features in terms of the probabilistic expression evaluation on the set of terminal states. We defined features of the groups solution

cardinality, distance and persistence which include all non-class-based features used in modern GGP agents. We integrated the features in our fuzzy evaluation function and could show that the features are indeed general and fast to construct, and that they significantly influence the playing strength of an agent.

Still, especially distance features posed a number of problems, most importantly their dependence on syntactic structures.

We addressed this problem by providing 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, preserves piece-dependent move patterns and produces an admissible distance heuristic.

We measured its efficiency and quality and found that is especifically useful if our standard (i.e., non-admissible) distance feature acquisition approach does not detect features. In these cases it increased the playing strength of the agent. Given the higher time requirements for constructing the feature, we suggested to use the algorithm as a complement to non-admissible feature detection that is used if no other distance feature could be found. Still, given its theoretical superiority and a number of promising extensions, we understand this work as a first step towards further research in this area.

4.6.1. Future Work

We hope to have provided some useful insights into how features can be best acquired and used in a General Game Playing agent. While our work seems certainly fruitful in the light of the evaluation results, many questions remained unanswered.

We discuss these for rule-derived and admissible distance features separately.

Rule-Derived Features

One problem is the mapping of feature output values to truth values of our aggregation function. Questions related are whether $\mathtt{f}_p$ and $\mathtt{t}_p$ are the only relevant values, what are better values and how are they best determined. While our mappings seem to some extent reasonable, we strongly doubt that they are optimal. We also found that the values $\mathtt{f}_p$ and $\mathtt{t}_p$ should differ across different formulas, however, we have no method at hand that provides us with specific values depending on the formula. Finally, we addressed the evolution of $\mathtt{f}_p$ and $\mathtt{t}_p$ towards $\mathtt{f}$ and $\mathtt{t}$ in a simplistic, linear manner. Again, while this seems practical, it should not be optimal and should as such be subject to further research.

A second issue becomes apparent when looking at the absolute dimension of features. If distance and solution cardinality features would provide us with an integer describing time steps, we could compare these time steps against our terminal estimation and derive better conclusions whether a formula with a distance interpretation is still reachable or a formula with solution cardinality interpretation can be guaranteed to hold in the

terminal state. While the necessary calculations with respect to the features are readily accessible, an estimation of how many time steps the match will last is lacking.

Admissible Distance Features

With respect to admissible distances we distinguish possibilities for future work by their effect on the distance features.

Increase Efficiency The main problem of the approach is its computational cost for constructing the fluent graph. The most expensive steps of the fluent graph construction are the grounding of the DNF formulas $\mathcal{N}$ and the subsequent processing of the resulting large formulas to select edges for the fluent graph. For some complex games, the algorithm takes more than 100 seconds. Thus, reducing the size of the formulas likely leads to faster construction.

One way to reduce the size of $\mathcal{N}$ is a more selective expansion of predicates (line 5) in Algorithm 1. Developing heuristics for this selection of predicates is one of the goals for future research.

In addition, it would be worthwhile to explore a way to construct fluent graphs from non-ground representations of the preconditions of a fluent to skip the grounding step completely. For example, the partial fluent graph in figure 4.11 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 would be exponentially smaller while still containing the same information.

Increase Generality The technique of constructing fluent graphs can be extended in many ways. Most importantly, it is possible to obtain a fluent graph of sufficient preconditions. In less complex games such as Tic-Tac-Toe this should be possible without major problems. Since each fluent then may have several sources, the distance from a state to a fluent would have to be defined as the maximum of all distances of the preconditions holding in current state towards the desired fluent. Preconditions for persistence could be obtained directly from the graph and there would be no need for similarity heuristics to select a precondition.

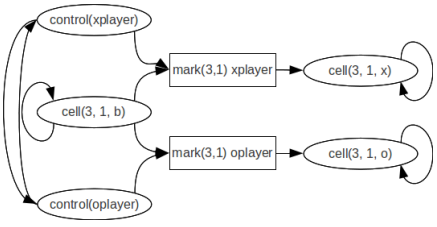


Figure 4.20.: Partial Fluent Graph for Sufficient Preconditions in Tic-Tac-Toe

Figure 4.20 shows such a fluent graph. For better readability, it includes besides the fluents also action nodes that indicate when an action is needed to obtain another fluent.

Taken one step further, the fluent graph could be generalized to arbitrary state-dependent formulas. In this case, a state can be represented as a mapping of each node in the graph to true or false. The distance of a state to a formula could be modeled as the sum (minimum) of the preconditions in case of a conjunction (disjunction). Since each formula ultimately depends only on fluents, we can reduce formula distances to fluent distances. For instance, the evaluation of the formula $row(3, x)$ could be modeled as the sum of the distances towards the three constituting fluents $cell(3, 1, x)$,$cell(3, 2, x)$ and $cell(3, 3, x)$.

Constructing such an formula graph should be easy in games like Tic-Tac-Toe where there is a small set of possible fluents and there are no additional difficulties such as negative preconditions. If it can be constructed, the (potentially perfect) evaluation function could be represented as a distance estimate of the state at hand towards the goal formula.

Apart from the aforementioned idea, formula graphs could also prove useful as representation to obtain transition probabilities for individual fluents / formulas.

5. General Evaluation

We conclude this thesis with a general evaluation on the playing strength of our agent. While the focus of the prior sections was on aspects of evaluation functions and their optimization, we now want to demonstrate that the approaches work together and produce convincing results.

In short, our hypothesis is that the approaches discussed within this thesis produce a competitive General Game Player. For an evaluation of this proposition we will analyze the outcomes of recent GGP competitions and pit our agent against a number of other successful GGP agents.

Before starting, we quickly specify the features used in the final version of Nexplayer.

5.1. Final Version of Nexplayer

The version of Nexplayer used for evaluation is also the final version of Nexplayer and unites all construction ideas and mechanisms discussed throughout this thesis.

These are generally

An Evaluation Function based on Neural Networks as discussed and evaluated in section 3.2 and enhanced in section 3.3.

Acquisition of Rule-Derived Features as discussed in the sections 4.3 and 4.4 and evaluated in section 4.4.4. All features were used and all improvements discussed in section 4.4.3 activated.

Acquisition of Admissible Distance Features as discussed in section 4.5 and evaluated in section 4.5.4. Due to the time requirements of the rule grounding and fluent graph construction, we applied the feature after we found that no simpler rule-derived distance feature could be used.

As search we used MiniMax and MaxN search and applied the following improvements:

- Transposition Tables using a Zobrist Hash [Zob70]
- alpha-beta pruning for MiniMax search [KM75]
- History Heuristics for MiniMax search [Sch89]

In the case of games with simultaneous moves, we applied paranoid Minimax.

In addition we used other improvements:

Compilation

Nexplayer uses ECLiPSe Prolog[ECL12] that allows for code generation and compilation at run time. This allows for compiling persistent facts into the evaluation function once they are known to persist.

The evaluation function construction process consists of the following separate stages:

Function Construction All data that represents the evaluation function (i.e., network structure, weights, ...) is generated and written into a persistent format.

Function Speedup On loading the evaluation function, data-sensitive parts of Nexplayer are specialized towards the data and recompiled.

Class-based Features

We also use a small number of class-based features as defined in section 4.1.2. These are

- Mobility
- Depth
- Piece Value
- Goal Value

Before applying mobility and goal value we confirm whether these can be calculated quickly. As threshold we used 0.3 seconds for 100 states. For piece value we determine the value of a piece by putting it on a blank board and determining the average number of moves it has to its avail.

Mobility and piece value are integrated as a bonus on the state value. The bonus is a small multiple of the machine precision and works thus as a tie-breaker. Depth is used in the same way but as a negative bonus to favor states closer to the current state.

Finally, Goal value is used for normalizing the heuristic value: If a goal value can be determined, then the state value is a linear combination of the goal value and the original value of our evaluation function.

Recursion and Chain Resolution

Recursions and chained conjunctions (conjunctions where conjunct i uniquely sets a variable that is used in conjunction $i + 1$) are often used to traverse a structure in the game and aggregate some value depending on the underlying fluents. An example for a chained conjunction would be the following statement:

```
1 count_stones(Red, Black) :-
2   count_stone(1, 1, 0, 0, Red1, Black1),
3   count_stone(1, 2, Red1, Black1, Red2, Black2),
4   ..,
5   count_stone(8, 8, Red62, Black62, Red63, Black63).
```

`count_stone(X, Y, RI, BI, RO, BO)` is here a predicate that checks the coordinates `X`, `Y` for a red or black stone and sets `BO=BI+1` and `RO=RI+1` in these cases. The complete statement thus traverses an 8x8 board and counts the number of red and black stones.

The problem with such a conjunction is that it may fail in non-terminal states if for some cell coordinate neither a red nor a black stone is found. In this case, an evaluation of the conjunction would provide no useful information. We address the problem by evaluating each conjunct separately and set `RI=RO` and `BI=BO` in case a conjunct `count_stone(X, Y, RI, BI, RO, BO)` fails.

Likewise, if a recursive predicate can be transformed to a chained conjunction, we apply the same mechanism. Note this is just a way to ensure that the predicate can be evaluated. The only game in the evaluation where such a feature was applied is `CephalopodMicro`.

5.2. Past Competitions

While themselves not too descriptive, the competitions provide an insight into the ranking of single agents. We will evaluate the last three bigger challenges. This not only indicates the strength of Nexplayer but also of its opponents we use for evaluation in the next section. Note that these competitions were held some time ago, so they do not necessarily reflect the playing strength of the same agent today.

In addition, we emphasize that in most competitions only a few games were played per agent and only a few matches were played per game and agent. Thus, there is a high degree of random influence on the results and the results should be taken with a grain of salt.

5.2.1. German GGP Competition 2010

In 2010, an inofficial GGP championship was organized by the University of Potsdam. Mostly student teams participated. We obtained our data from the site `http://www.ggp-potsdam.de/wiki/gggpc2010` (German) and the GGP server [Sch12].

Preliminaries	Agent	Rank	Total Points
	Nexplayer	1	331
	Fluxplayer	2	281
	...	...	...
Finals	Fluxplayer	1	390
	ATAX_test	2	375
	tut	3	300
	Centurio	4	245
	Nexplayer	5	225

Table 5.1.: German GGP Competition 2010 - Summary

The preliminaries consisted of two groups with five agents each. 12 matches were

played and Nexplayer ranked first in its group with 331 points, followed by Fluxplayer. Both agents advanced to the finals. The final group consisted of 5 players. 11 matches were played in 5 games. Nexplayer ranked fifth.

5.2.2. International GGP Competition at IJCAI 2011

The GGP championship consisted of several matches in 10 games. The tournament mode was double elimination, 12 agents participated. It is possible to derive a ranking using the time of elimination.

Agent	Rank
TurboTurtle	1
Cadiaplayer	2
Ary	3
Nexplayer	4

Table 5.2.: International GGP Competition 2011 - Summary

Table 5.2 reconstructs the ranking based on the information found at `http://games.stanford.edu/`. Other agents in the tournament were Fluxplayer, Turk, Atax, Gamer, Centurio, Hydra (withdrawn), Yggdrasil (withdrawn), Toss (withdrawn).

5.2.3. German Open 2011

There were eight agents participating at the German Open, organized by the university of Bremen. The tournament was held in the beginning of October 2011 and all information was taken from the corresponding website `http://www.tzi.de/~kissmann/ggp/go-ggp/` at March 23, 2012.

The preliminaries ran for two weeks prior to the competition. They included the approximately 200 games hosted on the GGP Server [Sch12].

Player	Played Matches	Avg. Reward
Cadiaplayer	853	60.790
Nexplayer	1079	57.374
Ary	1173	56.909
Fluxplayer	1159	56.420
Alloy	487	48.632

Table 5.3.: German Open Preliminaries

The actual competition consisted of two rounds. Eight agents participated. In the first round, two groups were formed consisting of agents with an even / odd rank in the preliminaries. 12 matches were played in four games and the top two agents of each group formed the final group. In the final group, 19 matches were played in five games and agent with the highest total points achieved declared the winner of the tournament.

Heats	Agent	Rank	Total Points
	Nexplayer	1	261
	Fluxplayer	2	223
	Centurio	3	75
	Hydra	4	58
Finals	Ary	1	267
	Alloy	2	191
	Fluxplayer	3	169
	Nexplayer	4	167

Table 5.4.: German Open 2011 - Summary

Nexplayer advanced to the finals as winner of its group with a total of 261 points. In the finals, Nexplayer ranked fourth with 167 points. The corresponding consolation group was led by Cadiaplayer, followed by Centurio.

5.3. Experiments

We finish with an experimental evaluation against the three agents with an agent program available at the time of March 15, 2012.

The agents are

Cadiaplayer tagged as version 2.0.1 of June 08, 2011, downloaded March 15, 2012 from `http://cadia.ru.is/wiki/public:cadiaplayer:main#cadiaplayer_source`

Centurio tagged as version 2.1 of October 2010, downloaded March 15, 2012 from `http://www.ggp-potsdam.de/wiki/Releases`

Fluxplayer Version as of January 31,2012. The agent is available upon request.

The agents are all tested and participated successfully in several tournaments. Specifically, Fluxplayer won the GGP Championship in 2006 and Cadiaplayer won the championship in 2007 and 2008.

We set up a series of matches of Nexplayer against each of these agents on a set of 28 test games for the two clock setups 20,4 and 40,8.

5.3.1. Centurio

We played 10 matches for each clock setup against Centurio, so 28x2x10=560 matches were played.

Figure 5.1 and Table 5.5 show the average result of the matches. We see that Nexplayer outperforms Centurio. In fact, there is only a handful of setups where Centurio achieves more points per match than Nexplayer. These are for the 20,4 clock the three checkers variants, `othello-comp2007` and `racer` and for the 40,8 clock setup `chinesecheckers2`,

Clock	Nexplayer	Centurio
20,4	70.3	32.7
40,8	82.2	23.3
Average	76.3	28.0

Table 5.5.: Average Points Nexplayer vs Centurio

`othello-comp2007` and `quarto`. Among these, the greatest margin was achieved in the three Checkers variants for the 20,4 time settings due to an incomplete network creation.

5.3.2. Cadiaplayer

Due to difficulties in running Cadiaplayer directly on our 64-bit system, we had to set up a Virtual Machine with a 32-bit version of Ubuntu 11.10. We equipped the VM with 2GB of RAM and 2 processors to allow for smooth operation.

We played 10 matches for each clock setup against Cadiaplayer with standard settings, i.e., using just one processor. All in all, 28x2x10=560 matches were played.

Clock	Nexplayer	Cadiaplayer
20,4	70.1	25.8
40,8	94.6	3.9
Average	82.3	14.9

Table 5.6.: Average Points Nexplayer vs Cadiaplayer

Figure 5.2 and Table 5.6 show the average result of the matches. We see that Nexplayer outperforms Cadiaplayer. Again, there are only few setups where Nexplayer achieves less points than Cadiaplayer. These occur all for the 20,4 time clock and are `bidding-tictactoe`, `chinesecheckers2`, `connect4`, `crisscross`, `pacman3p` and `racer`. The difference to the 40,8 clock setup suggests that these are related to incomplete feature detection.

Noteworthy is the game of `skirmish` where we thought an error occurred but found that both players were playing for minimal risk, with both agents resulting in 0 points. Hence a win rate can not be calculated.

5.3.3. Fluxplayer

We played 20 matches for the two clock setups against Fluxplayer. Since we know that unlike UCT agents Fluxplayer benefits from greater start clocks, we additionally add a 120,8 clock setup to the evaluation. This increases the likelihood that Fluxplayer uses the best evaluation function it can come up with. In total, 28x3x20=1680 matches were played.

Figure 5.3 and Table 5.3 show the results of the matches.

Obviously, Fluxplayer is much more competitive than its probabilistic counterparts. There are some games in which Nexplayer does not achieve points, such as `breakthrough`,

Clock	Nexplayer	Fluxplayer
20,4	57.02	45.39
40,8	60.14	44.38
120,8	55.70	49.66
Average	57.62	46.46

Table 5.7.: Average Points Nexplayer vs Fluxplayer

`breakthrough_suicide` for all clock setups or `racer` and the checkers variants for the 20 second start clock setup. In the Breakthrough variants Nexplayer cannot prove the game zero-sum and loses advanced pawns this way. In the Checkers variants with 20 seconds start clock the Prolog engine breaks for some reason which is why we also have no information on the number of states.

Despite these single failures, the experiments demonstrate that even without these points Nexplayer performs very well against Fluxplayer. The overall win rate is highest for the 40,8 clock setup and reduces for higher start clocks when Fluxplayer is able to catch up.

Figure 5.4 shows the number of unique states inspected per second. We can generally see that for a higher start clock Nexplayer becomes slower due to a higher number of features detected and applied. In most cases Nexplayer is slightly faster than Fluxplayer.

5.4. Summary

Given the tournament results we conclude that Nexplayer is a competitive agent. Specifically, the results of the preliminaries of the German Open 2011 and the matches against Fluxplayer demonstrate the quality and efficiency of our approaches that were the general goals for our thesis. Nevertheless, in any competition a good and stable performance was achieved, specifically, in the more recent tournaments where the admissible distance features and performance optimizations based on features were introduced.

The locally conducted experiments confirm the playing strength of Nexplayer.

That being said, there is a number of reasons why the experiments may be positively biased.

Old Versions We tested our player often enough in the given games to find all major bugs. In the same way, we had the time to consider all research contributions up to date and decide whether we want to include them. This type of intervention was not possible for the creators of other agents. One the one hand, they were released for download in 2010/2011 (Centurio, Cadiaplayer) and did have less time to improve existing or include new properties. On the other hand, while no agent failed visibly, it is possible that some minor errors resulted in a suboptimal decision.

Single Core Execution All agents ran using a single CPU core. Scalability therefore was no issue. Given the last few competition results, playing strength of proba-

bilistic agents draws mainly from its easy-to-apply thread-level and cluster-level parallelism.

Biased Games Our games were, with the exception of `pacman3p`, two-player games. However, probabilistic agents perform much better in multiplayer games (see section 3.1.1) due to the greater degree of uncertainty induced by the moves of other players.

No Optimization We tested each opponent agent as-is, without optimizing it.

While each of the above points should have a minor impact on the evaluation, the point of single core execution vs parallelism deserves attention. The absence of parallelism in the above experiments demonstrates a much more resource-efficient state evaluation by deterministic knowledge-based agents and highlights at the same time their major weakness. Still, Nexplayer is within striking range of its opponents and should be even harder to win against if parallelized.

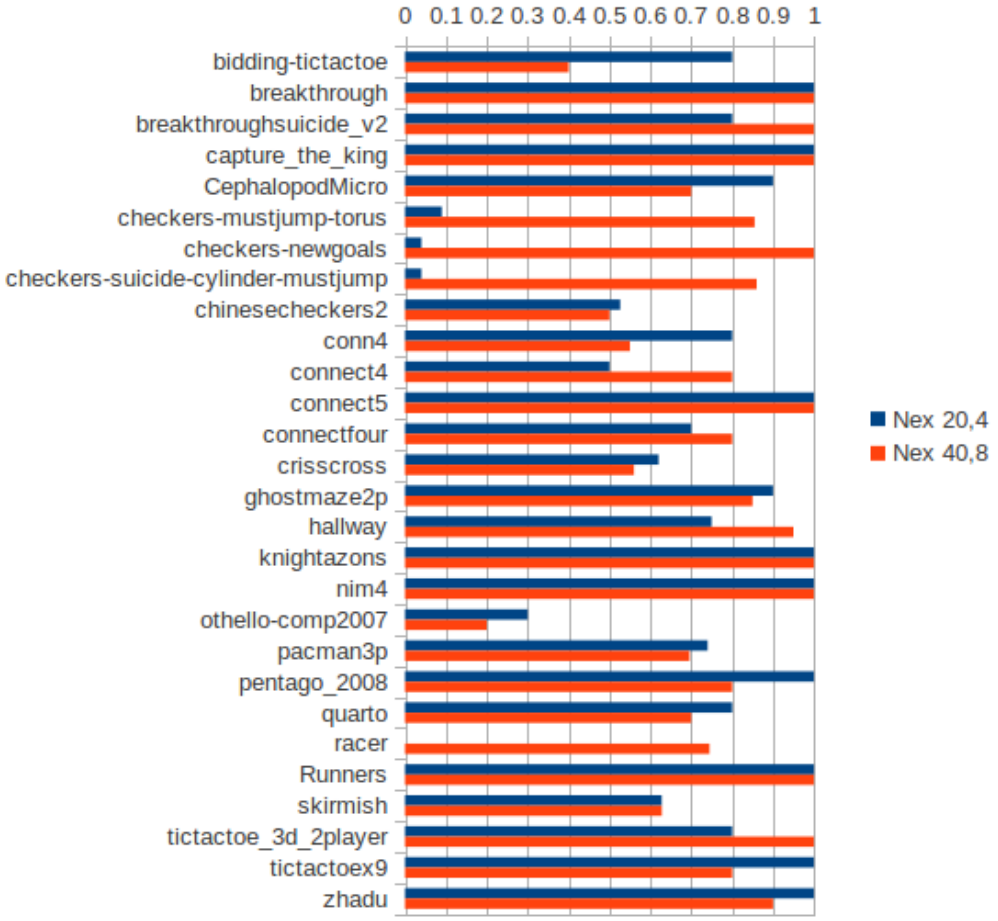


Figure 5.1.: Win Rate of Nexplayer vs Centurio

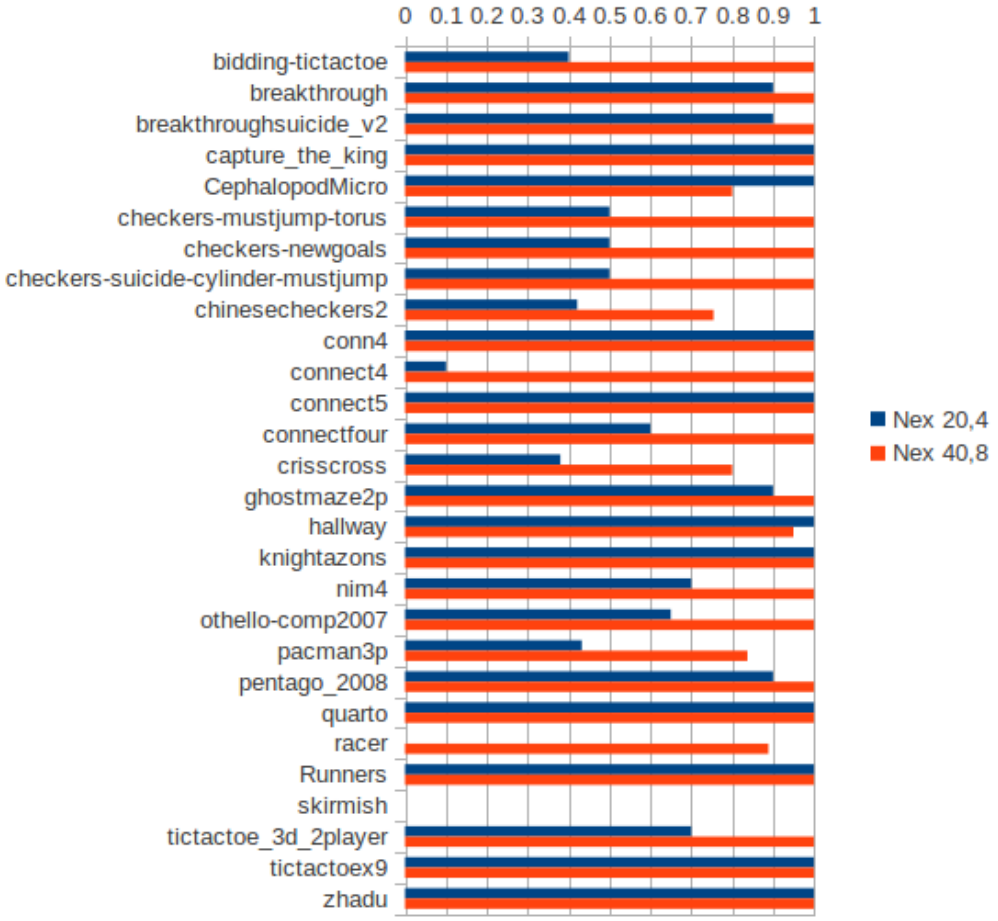


Figure 5.2.: Win Rate of Nexplayer vs Cadiaplayer

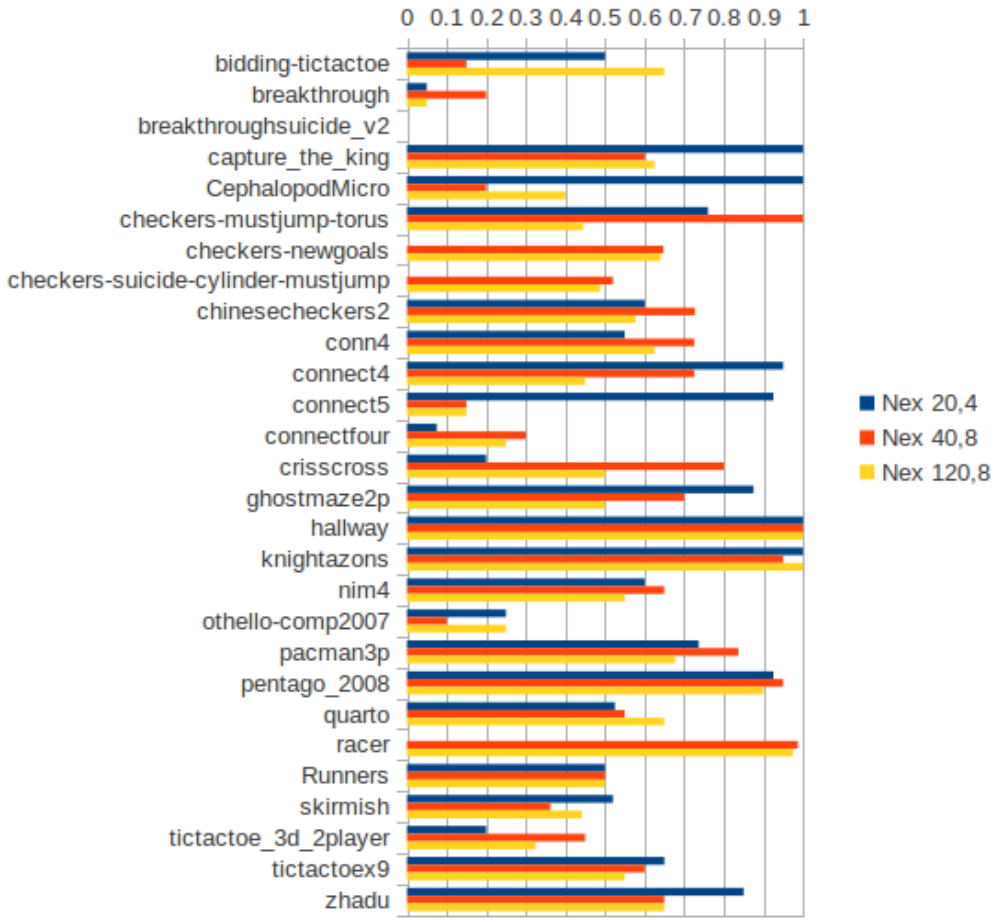


Figure 5.3.: Win Rate of Nexplayer against Fluxplayer

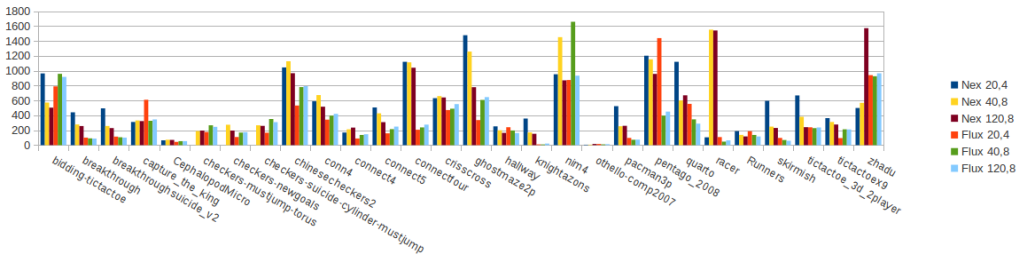


Figure 5.4.: Nexplayer vs Fluxplayer: States per Second

6. Related Work

This section is intended for discussing comparable approaches that develop agents that can play whole classes of games as opposed to a single fixed game.

We start by discussing probabilistic GGP agents in the next section, proceed with deterministic GGP agents and finally discuss older prominent predecessors.

6.1. Probabilistic Agents

Probabilistic agents employ UCT search and a probabilistic evaluation function. The agents differ mainly by their knowledge-based enhancements, their degree of optimization and their approach on parallelization.

Each probabilistic GGP agent represents a multi-agent system where each role in a game is represented by a "subagent" situated in a single-player game with a non-deterministic environment. Each subagent is unaware of the presence of other agents and simply chooses one of its legal actions. The choices of other agents are hidden and are modeled as non-deterministic response of the environment.

6.1.1. Cadiaplayer

Cadiaplayer [FB08] won the international GGP competition in 2007 and 2008 and is one of the most competitive GGP agents which could be seen in the preliminaries of the German Open.

In addition to standard UCT search equipped with a probabilistic evaluation, Cadiaplayer employs a number of extensions that turn their agent more knowledge-based [FB10] and mitigate the drawbacks of the probabilistic evaluation:

Rapid Value Estimation (RAVE) After playouts, the action values of siblings occurring in the UCT tree are also updated if the same action occurred in the playout. This provides first estimates on the values of actions that were only little explored. The score is kept separately and its influence on action selection is decreased with an increasing number of samples of the state [GS07].

Move-Average Sampling Technique (MAST) For each action performed during an iteration from a the root of the UCT tree to the terminal state the average playout value is kept. These action values are then used to bias the playout towards actions with a higher value. This introduces statistical knowledge over actions and results in more realistic playouts.

Tree-Only MAST (TO-MAST) Instead of updating action values for a complete episode as in MAST, only action values within the UCT tree are updated. This limits the number of samples for action values but increases the precision since only values of deliberately performed actions are updated.

Predicate-Average Sampling Technique (PAST) Similar to MAST, action values are kept, but actions are seen as depending on the occurrence of predicates in a state. Action values are updated in a state for all predicates in that state. For playouts the action is used that has the highest value for all the predicates in the state. This provides a context for actions and introduces statistical knowledge over actions depending on states.

Features-to-Action Sampling Technique (FAST) A basic set of features is detected (currently piece value and specific cell features) and after each playout the set of corresponding feature weights is updated. These weights are then used to increase the action value of actions in the playout. This introduces more complex knowledge based on features.

Following the authors, the extensions increase the strength of Cadiaplayer significantly.

6.1.2. Ary

Ary [MC11] won the international GGP competition in 2009 and 2010. While its exact details are unknown, it is known that Ary consists of a master-slave hierarchy among distributed program instances. The master is responsible for the outside communication and contains the UCT tree. All playouts of states are submitted to a slave. When the slave finished the evaluation, it returns the results to the master who adds the information to the tree. Due to the asynchronous organization of playout delegation and playout value retrieval, Ary is fast.

6.1.3. Centurio

Centurio [MSWS11] applies UCT with the RAVE extension. Its strength seems to be primarily obtained due to heavy optimization, using code optimization[Wau09] and parallelization techniques on the thread level (i.e., on a single machine) as well as on cluster level (i.e., across several machines).

6.1.4. Gamer

Gamer [KE11] applies UCT together with parallelization techniques on the thread and cluster level. The agent tries to ground-instantiate the rules in order to improve efficiency of the algorithm. If the rules can be ground-instantiated, additionally a solver is started that tries to solve the game by backpropagating the best moves from the terminal states to non-terminal states.

6.1.5. TurboTurtle

TurboTurtle is developed by Sam Schreiber and won the 2011 GGP competition. However, we currently have no further information to our avail other than the fact that it is a probabilistic agent.

6.2. Deterministic Agents

Deterministic agents evaluate features on states and aggregate their results to a state value.

6.2.1. Ogre

Ogre [Kai07a, Kai07b, Kai07c] is a deterministic agent that relies on a linear combination of features. As features, Ogre uses mostly class-based features (goal, mobility, depth, piece value) plus quantities and distances. The latter two as well as piece value seem to be based on the detection of boards. These boards are detected by playing random matches and identifying fluents with arguments that exhibit little variance across states.

For determining weights, we could not find an algorithm other than using the same weight for each feature.

With its reliance on random matches, Ogre does not exhibit the efficiency criteria needed. In addition, it seems its features are strongly class-based (board-based) features and thus tailored towards classes of games at the expense of generality.

6.2.2. Kuhlplayer/UTexas

Kuhlplayer [KDS06, Kuh10] identifies a set of game specific structures such as boards, counters, and pieces by matching them against syntactically predefined patterns.

A set of features is derived from the structures found according to the given table:

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

Along with these more sophisticated features, other features are created that work one the subset of (true) fluents in a state matched against a pattern, returning, e.g., the cardinality of that set. To determine the value of the features, they are plugged into a linear combination with weights +1 or -1 and ranked according to the difference in win rate they produce. If the difference is statistically significant, the feature is included in the evaluation function with a fixed weight, else it is discarded.

The final evaluation function thus consists of a linear combination of features with their absolute value depending on their rank and their sign depending on the correlation with the goal value.

The biggest problem of Kuhlplayer is the expensive construction of the evaluation function since it requires multiple matches to be played to evaluate a single candidate feature. The artificial setting of feature weights and the limited scope of feature detection impose further limits on the playing strength of the agent.

6.2.3. Cluneplayer

Cluneplayer[Clu07] won the international GGP competition in 2005 and uses an evaluation function consisting of three levels. The first level consists of the features that are combined into three high-level features of the second level. Finally, these are combined to a single state value.

Features

The starting point for the low-level features is the set of expressions appearing in the game description. Variables with only few possible instances are ground-instantiated and three possible interpretations are imposed on these expressions:

Solution Cardinality indicates the number of distinct solutions to an expression with variables.

Symbol Distance indicates the number of steps the fluents holding in the current state require towards the expression. This is achieved by seeing whether a fluent in the expression has a constant as argument that is part of a domain of a binary relation. In that case, the fluent is considered a goal. To determine the distance, the constant in the goal fluent is replaced by a variable and the resulting pattern identifies all possible sources for the goal fluent in the current state. Then the distance for each source fluent to the goal fluent is calculated based on the shortest path as defined by the binary relation. The corresponding article then states that the overall distance is the sum of the distances of each matching fluent in the current state towards the goal fluent. However, since the example given suggests that a minimum is used, it is possible that a sum (i.e., the Manhattan distance) is used for aggregating argument distances to a fluent distance and the minimum over all fluent distances is used as overall distance.

Partial Solution counts the number of fulfilled conjuncts in a conjunction.

The three given interpretations are applied on all expressions found and form a basic candidate feature set. This set is extended by relative features that, e.g., compare the solution cardinality of two expressions if both are symmetric in the initial state and appear an equal number of times in an equal number of rules in the game description. The candidate feature set is refined by removing all unstable features, that is, all features that wildly oscillate during a simulated match.

These features are used to model three higher-level features in the second level:

The *payoff* $P \in [0, 100]$ indicates the immediate payoff of a state if the state was terminal. It is modeled as a weighted sum where all features positively correlated to the goal values of states are weighted by their stability factor and all negatively correlated features are weighted by their negative stability factor. The stability is higher for more stable features, assigning the highest weights to the most stable features with. For the payoff feature only features occurring in the goal condition are used and if one feature subsumes another, only the subsuming feature is retained. For instance, if a comparison of two solution cardinalities is used for modeling payoff, then the two solution cardinality features are ignored for the payoff function.

The second high-level feature is control. *Control* $C \in [-1, 1]$ is defined as the normalized number of legal moves of the agent relative to the number of moves of his opponents. The series of control values can be calculated for an internally simulated match and this control value series is then modeled by applying least squares regression on the features. This time, only features that appear in the legality relation are considered and again, features subsumed by others are ignored.

Finally, *termination* $T \in [0, 1]$ is modeled also by least squares regression on the features occurring in the terminal rules.

Aggregation

These three high-level features are then aggregated as seen in equation (6.2.1).

The value of a state for a specific role is defined as

$$eval(s) = T(s) * P(s) - (1 - T(s))((50 * (1 + C(s)) * S_C + S_P * P(s)) \qquad (6.2.1)$$

Here, $S \in [1, \infty]$ is the stability of the payoff S_P, or control S_C and indicates to what extent the payoff / control remains stable ($S \to \infty$) or oscillates wildly ($S \approx 1$) during the course of a match. The purpose of the function is to first promote control and gradually favor payoff for higher termination values which in this case are interpreted as probabilities.

Summary

The evaluation is consistent and compact and has bounded computation costs. The cost of construction, however, is high due to empiric determination of feature parameters by simulation.

As Kuhlplayer, Cluneplayer cannot guarantee strict approximation or extrapolation of the goal function due to the specific evaluation function model.

However, there are significant improvements compared to Kuhlplayer with respect to each of these properties.

A good approximation of the perfect evaluation function is easier to find as Cluneplayer does not define a restricted set of syntactically detected features. Instead, it defines operations that can be used on arbitrary expressions in the game description to obtain such features. These features include the features detected by Kuhlplayer. Consequently,

the feature set is richer. In the same way, weights are not fixed but determined by stability of the feature and its correlation to goal the values of goal states.

Extrapolation is somewhat accounted for as the state value for a termination feature value of $T = 1$ equals the payoff P, that in turn is linked to the goal values.

Finally, function construction is faster since pseudo-terminal states are generated and subsequently used for training features. While this may have detrimental effects on quality, it effectively decouples evaluation function quality and construction time.

Nevertheless, we find the strength of Cluneplayer seriously limited by the unusual choices of the game model with parameters C, P and T. In our view, this greatly reduces the chances of approximating a perfect state value function due to a number of factors. Most important among them is the exaggerated importance of control.

6.2.4. Fluxplayer

Fluxplayer [ST07, Sch11] won the international GGP championship in 2006 and does, unlike all other deterministic agents, not rely on a linear combination of features as a evaluation function. Instead, it derives the evaluation function directly from the goal.

Aggregation

Given a role r, the agent transforms each goal rule $g(r, gv)$ to a fuzzy evaluation function $h(r, gv, s)$ and then combines the fuzzy values $h(r, gv, s) \in [0, 1]$ and the corresponding goal value gv to the state value $h(r, s)$:

$$h(r,s) = \frac{100}{\sum_{gv \in GV_r} gv} * \sum_{gv \in GV_r} gv * h(r, gv, s) \tag{6.2.2}$$

Each $h(r, gv, s)$ returns a value in the interval $[0, 1]$ where values closer to 1 represent a higher degree of truth. It is evaluated as

$$h(r, gv, s) = \begin{cases} eval(g(r, gv) \vee terminal, s) & : g(r, gv) \\ eval(g(r, gv) \wedge \neg terminal, s) & : \neg g(r, gv) \end{cases}$$

$g(r, gv)$ represents the unrolled goal condition, *terminal* the unrolled terminal condition and the function $eval(\epsilon, s)$ evaluates expression ϵ in state s via

$$\begin{aligned} eval(x \wedge y, s) &= fz_{\wedge}(eval(x, s), eval(y, s)) \\ eval(x \vee y, s) &= fz_{\vee}(eval(x, s), eval(y, s)) \\ eval(\neg x, s) &= fz_{\neg}(eval(x, s)) \\ eval(x, s) &= \begin{cases} p & x \text{ holds in } s \\ 1 - p & \text{else} \end{cases} \end{aligned}$$

The specific fuzzy functions $fz_{\wedge}$, $fz_{\vee}$ and $fz_{\neg}$ are consistent t-norm fuzzy functions of the Yager family where

$$\begin{aligned} fz_{\neg}(x) &= 1 - x \\ fz_{\vee}(x, y) &= \sqrt[q]{x^q + y^q} \\ fz_{\wedge}(x, y) &= fz_{\neg}(fz_{\vee}(fz_{\neg}(x), fz_{\neg}(y))) \text{ (de Morgan)} \end{aligned}$$

The parameters p and q are set to 0.75 and 15

Evaluations of the type $eval(x, s)$ where x is a fact (i.e., it is not a conjunction, disjunction or negation) are then further improved by substituting them with features.

Features

Before applying features, Fluxplayer runs a domain analysis that detects the following general properties:

Order Predicate is a binary, static, antisymmetric and transitive predicate.

Successor Predicate is a binary, static, antisymmetric, functional and injective predicate.

Input and Output Arguments are the arguments of a fluent such that for each tuple of input arguments there is exactly one tuple of output arguments.

Fluxplayer then transforms fluents with ordered arguments to distance features. An ordered argument is the argument of a fluent whose domain is ordered. Since the transitive closure of a successor predicate is an order predicate, arguments with a successor relation are also ordered.

The features then are

Order Features that evaluate an order predicate based on the distance between its arguments

Quantities that are distances on ordered fluent output arguments

Distance Feature that are distances on ordered fluent input arguments

Besides the heuristics mentioned in the article, we know of personal contact that a basic mobility representing the number of moves available in a state and a piece value estimation representing a material value based on the overall-mobility (determined by simulation) of a piece is added to the state value if applicable. Also, persistence information is used.

Property	Probab. Agents	Kuhl, Ogre	Clune	Flux
Efficiency				
Compactness	+	+	+	+
Bounded Computation Cost	-	+	+	+
Ad-Hoc Construction	+	-	(-)	+
Quality				
Consistency	(-)[a]	+	+	+
Strict Monotonicity	?[b]	-	-	?[c]
Extrapolation	+	-	(-)	+

[a] holds for repeated evaluations
[b] holds for specific games
[c] holds for descriptive goal functions

Table 6.1.: Categorization of the Aggregation Functions of GGP agents

Summary

Fluxplayer uses an elegant solution for the construction of an evaluation function. Besides being compact and consistent, its construction time is bounded by the size of the goal description, making construction fast.

From a qualitative viewpoint, the evaluation function represents an extrapolation of the goal function from terminal states to non-terminal states.

Moreover, the evaluation function is strictly monotonic and thus allows for the detection of symmetric states if the symmetry is expressed in the state and goal condition. On the other hand, if the goal function is not descriptive or is structured in a way that misrepresents the perfect state evaluation, the monotonicity of the evaluation function is not aligned to that required by the perfect evaluation.

With respect to features, Fluxplayer uses a feature detection focused on expressions in the goal function.

We conclude that all basic qualitative properties hold *if* the goal condition reflects the valuation. The only disadvantage of Fluxplayer is its fixed function form that does not allow for learning or changing weights.

6.3. Summary of GGP Agents

Table 6.1 summarizes the main criteria for the evaluation functions of current agents as discussed in section 2.5.

While most criteria are self-explaining, a quick word on strict monotonicity is necessary: For probabilistic agents, the evaluation function is technically never strictly monotonic since it is not consistent. Nevertheless the values of states are stabilized and aggregated via UCT and can be considered closer to fulfilling the monotonicity criterion for nodes in the UCT tree that are close to terminal states and for games where the random move assumption is valid.

Note that in addition to the above criteria, no agent allows for learning , i.e., none fulfills the requirement as mentioned in section 3.1.3. A reason is the narrow time constraints of a few minutes start clock as used in the GGP tournaments.

6.4. Non-GGP Approaches

A small number of approaches on the automatic detection and construction of features has been presented that, to some degree, can be reused in the field of GGP. We will describe these processes swiftly and limited to the extent useful for feature detection and construction. As a result, the following descriptions may be incorrect in special cases that we do not discuss, however, the description is sufficient and consistent to grasp the main ideas of the approach. For a more detailed description of the systems we recommend reading the indicated literature, given that we find it not only useful in the context of this work, but also interesting.

6.4.1. Zenith

A general scheme of automatically constructing features was applied in the semi-general game playing agent Zenith [Faw93]. A feature in Zenith consists of a First-Order Logic formula and its output is the number of solutions to the formula in a state. Zenith operates on a set of features that is constantly enhanced by features derived from the set elements. It starts with a set containing only the goal condition of the game and per round extends the set of features by the results of the following four basic transformations on each element of the set.

Decomposition A conjunction or an arithmetic comparison or calculation is split into its operands, or a negation is removed.

Goal Regression Using the domain theory, the preconditions are derived that are necessary for the feature to hold in the next turn.

Abstraction A variable is considered existentially quantified over the set of solutions to the formula, thus the number of solutions does not depend on the instances of the variable. Alternatively, the least constraining term is removed.

Specialization Disjuncts are removed, a recursive call replaced by its base case or a variable instantiated to values such that there is a solution to the formula.

The evaluation function after each round is then constructed as a linear combination of all elements in the feature set that are not too expensive to compute. Feature weights are then learned and the features that contribute the lowest accuracy to the evaluation (determined on a test set) are continuously removed from the function until the evaluation does not exceed a specified time limit.

The advantage of this feature construction approach is that features are constructed by directed search in a bounded feature space. The derived features correspond directly to the goal condition and as such seem promising to use in an evaluation function.

Each feature is justified by representing a trade-off between cost and evaluation function quality. However, by relearning feature weights and determining feature accuracy for each round, the part of selecting features for the evaluation function is expensive.

6.4.2. Metagamer

Metagamer [Pel93] is another semi-general game playing agent specialized on playing "symmetric chess-like games". As such, all games are alternating two-player games on a structure similar to a chess board on which pieces are placed. For this reason, the majority of features for evaluating a state in a reasonable way is already known and can hence be predefined.

In Metagamer, four different feature types are used:

Mobility to determine the current and overall number of moves a piece has to its avail

Threats to determine the number of captures that are principally possible or useful

Goal Distance to determine the distance of pieces to specific locations or the distance of a player to goal of capturing a set of enemy pieces

Material to describe the value of a piece as a function of its basic mobility, its possibility to promote and other predefined properties

These feature types are represented by so called advisors that capture a set of instances of the feature type. For example, mobility would be a feature type that consists of an overall mobility not depending on the state, a current mobility depending on the state with, e.g., other pieces possibly hampering, a value indicating the average number of moves to reach other cells on the board and so on. The output of these advisors are then linearly combined to form the evaluation function.

Although designed for a class of games, Metagamer relies heavily on standard features. These features depend entirely on the existence of pieces and can be applied because they are known to be meaningful in the class of Chess-like games. The choice of which advisors (features types) to use and what weight to assign them, however, is mostly made by the author. As a result, Metagamer can be considered a standard agent whose repertoire of playable games has been extended to those games playable with a specific predefined piece-centered feature set.

While ex-ante feature definitions do generally not combine with GGP, it shows that there are features that may prove valuable as soon as some properties of symmetric chess-like games, such as pieces placed on boards, hold.

7. Discussion

7.1. Summary

The goal of this thesis was the creation of an evaluation function such that

- construction of the state evaluation function is fast
- evaluation of the state evaluation function is fast
- the evaluation function provides good results

In the light of these criteria the main contributions of our thesis are:

Construction of an Aggregation Function The aggregation function proposed by us is based on neural networks. It can be derived directly from the goal rules of a game and can thus be constructed much faster than traditional linear combinations of features. Its evaluation is much faster than that of probabilistic agents as it does not depend on the depth of the game tree and provides consistent results. Finally, it is general enough to be able to capture the perfect evaluation function of a game and allows for training. However, unlike most trainable functions, it does not require prior training and can be instead initialized with a domain theory. As a side effect, an interesting solution for the integration of logic and learning systems is put into practice.

Improved Aggregation Function The first algorithm for the initializing the evaluation function ($C - IL^2P$) is not well suited for complex domains due to machine precision. We presented a modified way of initializing the neural network so as to preserve precision and, consequently, the strict monotonicity property of the evaluation function.

Rule-Derived Feature Acquisition The feature acquisition method proposed derives features from rules. In contrast to standard feature acquisition mechanisms, the acquisition time is faster by orders of magnitude since the set of candidate features is restricted to those that can be derived from expressions in the goal rules and there is no necessity for rating or weighting a feature in simulations. Still the acquisition is comprehensive as it interprets underlying expressions in different ways and covers all major features used in current GGP agents.

Acquisition of Admissible Distance Features The distance feature introduced first suffers from a number of disadvantages, namely its dependence on syntactic circumstances and its disregard with respect to the move pattern of the object involved.

We proposed a new way to address this problem by constructing and evaluating fluent graphs. Despite their costly construction, they are useful in our Game Player and, more importantly, are a first step towards distances between expressions. Such distances would allow for a unified treatment of features and potentially even solve a complete class of games altogether.

Comprehensive Evaluation The comprehensive evaluation supports the above claims and shows the strength of our deterministic, knowledge-based agent when compared on an equal basis against other agents. At the same time, it shows the stark contrast to the competition settings where a better machinery significantly improves the playing strength for probabilistic agents. The results indicate that Nexplayer is currently one of the most efficient GGP agent in terms of playing strength per CPU cycle.

Furthermore, we believe to have provided some insights into other areas that deserve attention, such as,

- initialization of neural networks based on a domain theory (as opposed to plain learning approaches),
- meaning of features,
- design of aggregation functions to preserve the meaning of features and
- possible ways of combining deterministic and probabilistic evaluation functions.

Most of these subject also allow for interesting areas for further research.

7.2. Future Work

One of the most important problems for evaluation functions is the piecemeal approach to feature acquisition. While each feature itself has a meaning, it is next to impossible to compare the output of two features and draw conclusions as to what feature represents a higher probability that its underlying expression becomes true or false. We did a first step in this thesis by explicitly showing that features represent evolution patterns of expressions and reducing features to a probability that the expression holds in the terminal state.

Still we are only at the beginning: We had to rely on what can be described at best as an informed guess in order to estimate the probabilities of the expression holding in the terminal state and strongly suggest that an evolution graph of expressions would already help to estimate their stability. The evolution graph would be similar to a fluent graph but edges would be labeled with transition probabilities. Such a graph would itself represent a reduced game with a few fluents and the corresponding transition probabilities could possibly be approximated by statistically aggregating the data from random playouts.

In a next step, the probabilistic data should be evaluated by an appropriate aggregation function. Even though Fluxplayer's fuzzy evaluation function does this and Nexplayer's is even adaptive, both do not quite seem to fit to the problem. Specifically it is unclear why the probability of an expression holding in a terminal state should be evaluated by a fuzzified goal condition and why this provides a good guidance when traversing the game tree. For now, fuzzy evaluation seems convenient (due to fuzzy logic and probability theory operating on the unit interval) rather than specifically plausible.

A strongly related issue is how probabilistic and deterministic evaluation functions combine. Following our view on features, a Monte Carlo playout is a mere sampling of a feature derived from the goal rule. The transition probabilities are sampled due to a lack of other viable approaches. This means that if there is another way to obtain the probabilities without sampling, Monte Carlo playouts could be rendered obsolete. On the other hand, deterministic functions may require playout data to obtain reliable information on the transition probabilities of expressions. Both points should be investigated and we are confident that both approaches are possible and lead to a new understanding of evaluation functions and classes of games.

From a practical perspective we think it is mandatory to parallelize Nexplayer. Given the efficiency of Nexplayer, a simple Master-Slave model with a core program submitting state evaluation to slaves could speed up Nexplayer linearly and should make a noticeable difference.

Also, despite the current GGP competition settings that effectively discourage learning, we believe that there is a future for learning. Given that achieving mastership in a game takes a human years if not decades, it seems naive to believe that a computer could do this in minutes, at least for now. Since our evaluation function construction demonstrates how to initialize a neural network without taking the risk of bad initialization or non-convergence, we have a new tool for answering the question whether a learning evaluation function may converge towards the perfect evaluation function. In comparison to other approaches, odds seem certainly much higher since our network is not wildly grown but strictly corresponds to a propositional domain theory that can be used for supervision and tracing of the learning process by humans. Particularly, we are inspired by the findings of bootstrapping possibilities [VSUB09]. Still, we hesitate to apply learning since the last years of experience pushed us towards the belief that there is far more written in the game rules than meets the eye and that our problem is that we do not know how to extract this knowledge.

On a side note, we found that the search algorithms MaxN and paranoid MiniMax are much less suited for games with more than two players due to the inpredictability of the opponent's actions. Nexplayer seems to overestimate opponents and consequently plays too cautiously, silently assuming a coalition of its opponents. We believe this is a problem that another search algorithm could solve and our favorite candidate is UCT itself due to its ability to approximate mixed-strategy Nash equilibria.

7.3. Publications

Various parts of this work were published in the following peer-reviewed workshops and conferences.

- Using an aggregation function based on Neural Networks was first published in *"Neural Networks for State Evaluation in General Game Playing"*, European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), 2009 [MT09]. The conference is considered one of the leading conferences on machine learning and knowledge discovery.

- The follow-up work on the resolution problem was discussed in *"Neural Networks for High-Resolution State Evaluation in General Game Playing"*, International Workshop on General Intelligence in Game-Playing (GIGA), 2011 [Mic11b]. The workshop is held together with the International Joint Conference on Artificial Intelligence and is the main international venue specifically dedicated to General Game Playing.

- The detection of rule-derived features was discussed in *"Heuristic Interpretation of Predicate Logic Expressions in General Game Playing"*, International Workshop on General Intelligence in Game-Playing (GIGA), 2011 [Mic11a]

- The detection of admissible distance features was first discussed in *"Distance Features for General Game Playing"*, International Workshop on General Intelligence in Game-Playing (GIGA), 2011 [Mic11a] and published as *"Distance Features for General Game Playing"*, International Conference on Agents and Artificial Intelligence (ICAART), 2012 [MS12].

A. Appendix

A.1. Evaluation Setup

A.1.1. Environment

For evaluation we ran the agent(s) on the following machine:

- AMD Fusion A8-3850 (4x2.9GHz)
- 8 GB RAM

As Operating System we used Ubuntu 11.10 (64-bit). We used ECLiPSe Prolog Version 6.0 #189 [ECL12] for running the core of Nexplayer.

A.1.2. Games

For evaluation we selected only games that

- feature two roles or three roles with a minor impact of the third roles on the game
- cannot be searched exhaustively by Nexplayer

For evaluation we divided all games into categories that fit to the subject of evaluation. Each category represents a set of games where specific evaluation function properties are required to provide good results. However, these properties are not sufficient and consequently the categories are not mutually exclusive.

The main purpose of these categories is to provide evidence that the corresponding evaluation function property is useful and implementing it improves playing strength.

The categories are:

Structural Games are games with non-movable board elements and a complex goal function consisting of several conjunctions and disjunctions on a number of fluents. Required evaluation function properties are a fuzzy evaluation, a high resolution, persistence properties and to a smaller extent solution cardinality features.

Distance Games are games with movable elements and without captures. Required evaluation function properties are distance features.

Piece Games are games with movable elements and captures. Required evaluation function features are mobility, piece value and distance.

Other Games are other interesting games that do not fit in the above categories.

A.1.3. List of Games

All games referred to in this work can be found at the site of the GGP server [Sch12]. For evaluation, we specifically used the games with the following name:

Structural Games

- bidding-tictactoe
- conn4
- connect4
- connectfour
- connect5
- pentago_2008
- tictactoe_3d_2player
- tictactoex9

Distance Games

- chinesecheckers2
- crisscross
- ghostmaze2p
- hallway
- pacman3p
- racer
- nim4
- Runners

Other Games

- CephalopodMicro
- othello-comp2007
- quarto
- zhadu

Piece Games

- breakthrough
- breakthroughsuicide_v2
- capture_the_king
- checkers-newgoals
- checkers-suicide-cylinder-mustjump
- checkers-mustjump-torus
- knightazons
- skirmish

A.1.4. Opponents

As opponents we used the following agents:

Fluxplayer Version as of January 31, 2012

Cadiaplayer tagged as version 2.0.1 of June 08, 2011, downloaded March 15, 2012 from `http://cadia.ru.is/wiki/public:cadiaplayer:main#cadiaplayer_source`

Centurio tagged as version 2.1 of October 2010, downloaded March 15, 2012 from `http://www.ggp-potsdam.de/wiki/Releases`

Each agent was used in its default settings and was run on a single CPU core.

A.2. Other Improvements

The following are other improvements used that were not discussed in this work.

A.2.1. Delta Cache

Nexplayer also caches intermediate results of the evaluation function. These are only recalculated if the state under inspection is different with respect to the intermediate result.

In other words, for each state evaluation, the evaluation function first determines the delta between the current and the last state and then only recalculates the parts of the evaluation function that operate on the fluents that occur in the delta.

Due to the overhead we apply the delta cache only if we can guarantee that the average delta between two states affects in average less than half the leafs in the evaluation function.

A.2.2. Exclusion of Distances

In order to prevent false positives in the detection of features, we applied two enhancements for distance features.

Distances for Complete Sets of Fluents

A common instance to false positive distances are distances on a board. For instance, in some Connect Four variants disks drop to the bottom, thus it is probable that a rule

```
1 next(cell(X, Y2, red)) :-
2   true(cell(X, Y1, red)),
3   succ(Y1, Y2),
4   does(red, drop(X)).
```

exists. This would trigger our distance feature detection mechanism which we avoid by checking whether all possible instances that the fluent could evolve to are part of the goal condition. If confirmed, we skip the application of the distance feature.

Negative Distances

Negative distances pose a problem since they are incorrectly interpreted by our features. The standard distance feature for a role r correlates with the lower bound of moves needed to bring the distance to 0. The negation of such a feature promotes a high lower bound, that is, it favors states where a high number of moves is needed to reach the goal.

However, there are cases where this is not desired. An example is the game Runners. Here the player moves in a one-dimensional grid and wins if he walks from his starting cell 50 to cell 0. He loses, however, if he reaches cell -1 or -2.

The distance feature on cell 0 thus works as an attractor while the negated distances on cells -1 and -2 push the agent away. Consequently, the agent avoids coming close to cell 0 due to the perceived risk of reaching cell -1 and -2.

The problem is that the negated distance feature asks the question “How long does it take at most to reach the expression” while it should ask “How long can reaching the

expression be avoided?". Put more formally, it favors a high lower bound of the number of moves while we would be interested in a high upper bound of moves.

Both interpretations may coincide for forced moves, however, for unforced moves the upper bound is infinite. To express the latter we use mobility as a proxy: The higher the mobility of a role, the lower the probability that a move is forced.

Consequently, we substitute a negated distance feature occurring in the winning condition of a role r by the mobility of r if the underlying expression can be evolved by r. We assume that an expression can be evolved by a role if there is a move that affects the expression such that is does not hold anymore, but a similar expression holds in the next state.

A.2.3. Order of Feature Detection

The detection rules for the features presented are sensitive as to where they are placed. For example, it is necessary to enable detection of the solution cardinality for expressions only after ground-instantiation, as otherwise all non-ground expression would be identified as cardinality feature. On the other hand, solution cardinality can also be applied on non-ground fluents. However, in this case we do not want the fluent to be ground-instantiated since this bloats the evaluation function and generates no additional information.

Figure A.1 summarizes the control flow of identifying an expression. The detection of standard expressions is represented by the white oval nodes. Feature detection rules are depicted as gray rectangles. The control flow graph is divided into two parts. The left part shows the standard procedure for expanding expressions, the right side represents the complementing feature detection. There are two features side by side to emphasize that for expression truth as well as expression falsity features can be identified.

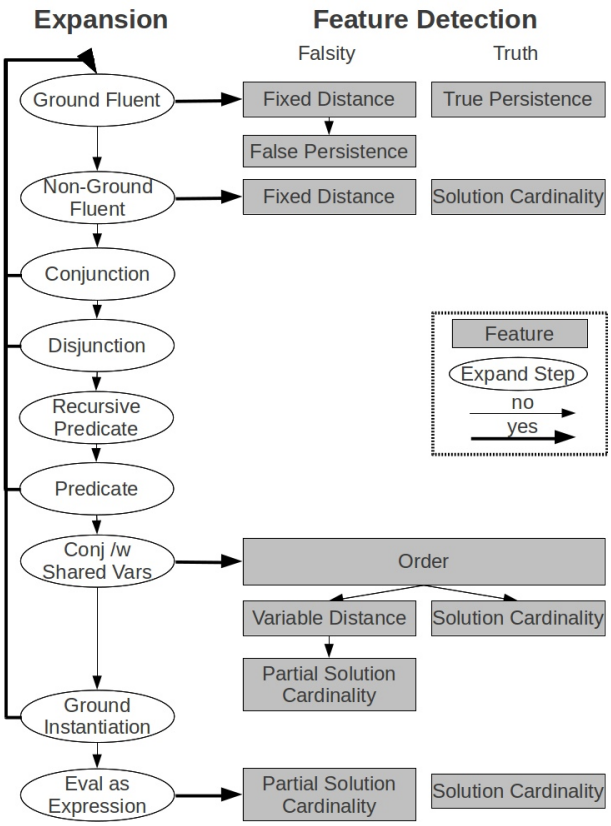


Figure A.1.: Feature Detection Graph

B. Bibliography

[Bad09] Sebastian Bader. *Neural-Symbolic Integration.* PhD thesis, Dresden University of Technology, October 2009.

[BH05] Sebastian Bader and Pascal Hitzler. P.: Dimensions of neural-symbolic integration - a structured survey. In *We Will Show Them: Essays in Honour of Dov Gabbay*, pages 167–194. College Publications, 2005.

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Inc., New York, NY, USA, 1995.

[Bur02] Michael Buro. The evolution of strong othello programs. In Ryohei Nakatsu and Jun'ichi Hoshino, editors, *Proceedings of the IFIP International Workshop on Entertainment Computing (IWEC '02)*, volume 240 of *IFIP Conference Proceedings*, pages 81–88. Kluwer, 2002.

[CJhH02] Murray Campbell, A.Joseph Hoane Jr., and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57 – 83, 2002.

[Clu07] James Clune. Heuristic evaluation functions for general game playing. In *Proceedings of the 22nd AAAI conference on Artificial Intelligence (AAAI-07)*, pages 1134–1139. AAAI Press, 2007.

[dCLd96] A.S. Zaverucha G. de Carvalho L.A.V. d'Avila Garcez. Logical inference and inductive learning in artificial neural networks. In *ECAI 96 Workshop on Neural Networks and Structured Knowledge*, 1996.

[dG02] A.S. Broda K. B. d'Avila Garcez and D.M. Gabbay. *Neural-Symbolic Learning Systems.* Springer, 2002.

[ECL12] The ECLiPSe Constraint Programming System. `http://eclipse-clp.org`, 2012. [Online; accessed April 13, 2012].

[Faw93] Tom E. Fawcett. *Feature Discovery for Problem Solving Systems.* PhD thesis, University of Massachusetts, Amherst, May 1993.

[FB08] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 259–264. AAAI Press, 2008.

[FB10] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game-playing agents. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 954–959. AAAI Press, 2010.

[GLP05] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62–72, 2005.

[GS07] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In Zoubin Ghahramani, editor, *Proceedings of the 24th International Conference on Machine Learning (ICML 2007)*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280. ACM, 2007.

[Gün08] Martin Günther. Automatic Feature Construction for General Game Playing. Master's thesis, Dresden University of Technology, September 2008.

[GWMT06] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA, 2006.

[HK94] Steffen Hölldobler and Yvonne Kalinke. Towards a massively parallel computational model for logic programming. In *ECAI'94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.

[HT10] Sebastian Haufe and Michael Thielscher. Pushing the envelope: General game players prove theorems. In J. Li, editor, *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, volume 6464 of *LNCS*, pages 1–10, Adelaide, December 2010. Springer.

[JS91] Mike McGann John Stanback. GNU Chess Heuristics. `http://alumni.imsa.edu/~stendahl/comp/txt/gnuchess.txt`, 1991. [Online; accessed March 15, 2012].

[Kai07a] David M. Kaiser. Automatic feature extraction for autonomous general game playing agents. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 93:1–93:7, New York, NY, USA, 2007. ACM.

[Kai07b] David M. Kaiser. The design and implementation of a successful general game playing agent. In *Florida AI Research Society Conference*, pages 110–115. AAAI Press, 2007.

[Kai07c] David M. Kaiser. *The structure of games*. PhD thesis, Florida International University, 2007.

[KDS06] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic Heuristic Construction in a Complete General Game Player. In *Proceedings of the 21st Nationral Conference on Artificial Intelligence (AAAI-06)*, pages 1457–62, Boston, Massachusetts, USA, July 2006. AAAI Press.

[KE11] Peter Kissmann and Stefan Edelkamp. Gamer, a general game playing agent. *KI*, 25(1):49–52, 2011.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[KPKP90] John F. Kolen, Jordan B. Pollack, John F. Kolen, and Jordan B. Pollack. Back propagation is sensitive to initial conditions. In *Complex Systems*, pages 860–867. Morgan Kaufmann, 1990.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[Kuh10] Gregory Kuhlmann. *Automated Domain Analysis and Transfer Learning in General Game Playing.* PhD thesis, University of Texas, 2010.

[LHH+08] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical Report March 4, Stanford University, Stanford, CA, 2008. The most recent version should be available at http://games.stanford.edu/.

[Llo87] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.

[MC11] Jean Méhat and Tristan Cazenave. Tree parallelization of ary on a cluster. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, pages 39–43, 2011.

[Mic11a] Daniel Michulke. Heuristic interpretation of predicate logic expressions in general game playing. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, pages 61–68, 2011.

[Mic11b] Daniel Michulke. Neural networks for high-resolution state evaluation in general game playing. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, pages 31–37, 2011.

[MS11] Daniel Michulke and Stephan Schiffel. Distance features for general game playing. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, pages 7–14, 2011.

[MS12] Daniel Michulke and Stephan Schiffel. Distance features for general game playing. In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence (ICAART-12)*, 2012.

[MSWS11] Maximilian Möller, Marius Thomas Schneider, Martin Wegner, and Torsten Schaub. Centurio, a general game player: Parallel, java- and asp-based. *KI*, 25(1):17–24, 2011.

[MT09] Daniel Michulke and Michael Thielscher. 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*, pages 95–110, Berlin, Heidelberg, 2009. Springer-Verlag.

[Pel93] Barney Pell. *Strategy generation and evaluation for meta-game playing.* PhD thesis, University of Cambridge, 1993.

[SBB+05] Jonathan Schaeffer, Yngvi Björnsson, Neil Burch, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Solving checkers. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 292–297. Professional Book Center, 2005.

[Sch89] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[Sch11] Stephan Schiffel. *Knowledge-Based General Game Playing.* PhD thesis, Dresden University of Technology, June 2011.

[Sch12] Stephan Schiffel. GGP Server. `http://ggpserver.general-game-playing.de`, 2012. [Online; accessed March 15, 2012].

[Sha50] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(4):256–275, 1950.

[Ski53] B.F. Skinner. *Science and human behavior.* Free Press paperback. Macmillan, 1953.

[ST07] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1191–1196, Vancouver, July 2007. AAAI Press.

[ST10] Stephan Schiffel and Michael Thielscher. A multiagent semantics for the game description language. In J. Filipe, A. Fred, and B. Sharp, editors, *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*, pages 44–55. Springer, 2010.

[Tes95] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[TF97] G. Thimm and E. Fiesler. High-order and multilayer perceptron initialization. *IEEE Transactions on Neural Networks*, 8(2):249–259, March 1997.

[Thi10] Michael Thielscher. A general game description language for incomplete information games. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 994–999, Atlanta, July 2010. AAAI Press.

[TS94] Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1-2):119–165, 1994.

[TSN90] Geofrey G. Towell, Jude W. Shavlik, and Michael O. Noordenier. Refinement of approximate domain theories by knowledge based neural network. In *Proceedings of the th National Conference on Artificial Intelligence (AAAI-90)*, volume 2, pages 861–866, 1990.

[TV10] Michael Thielscher and Sebastian Voigt. A temporal proof system for general game playing. In Maria Fox and David Poole, editors, *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 1000–1005, Atlanta, July 2010. AAAI Press.

[VSUB09] Joel Veness, David Silver, William T. B. Uther, and Alan Blair. Bootstrapping from game tree search. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1937–1945. Curran Associates, Inc., 2009.

[Wau09] Kevin Waugh. Faster state manipulation in general games using generated code. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, 2009.

[Zob70] Albert L. Zobrist. A hashing method with applications for game playing, 1970. Technical Report 88, Computer Sciences Department, University of Wisconsin.