

**Diplomarbeit**

Automatic Feature Construction  
for General Game Playing

by

**Martin Günther**

born on November 10, 1980 in Berlin-Neukölln

Dresden University of Technology

Department of Computer Science

Artificial Intelligence Institute

Computational Logic Group

Supervised by:

**Prof. Dr. rer. nat. habil. Michael Thielscher**

**Dipl.-Inf. Stephan Schiffel**

Submitted on October 22, 2008

**Günther, Martin**

Automatic Feature Construction  
for General Game Playing

Diplomarbeit, Department of Computer Science  
Dresden University of Technology, October 2008

## Aufgabenstellung Diplomarbeit

Name, Vorname: Günther, Martin  
Studiengang: Informatik  
Matrikelnummer: 2849363  
Thema: ***Automatic Feature Construction for General Game Playing***  
Zielstellung: Das *General Game Playing (GGP)* ist ein junger Forschungszweig, dessen Herausforderung darin besteht, universelle Spielprogramme zu entwerfen. Diese *General Game Player* sollen in der Lage sein, ein beliebiges Spiel zu beherrschen, von dem sie nur die Regeln in Form von Axiomen erhalten. Ein wesentliches Problem besteht darin, aus diesen Regeln eine Evaluationsfunktion zur Nutzung in der Spielbaumsuche abzuleiten. Der Ansatz von Fawcett (1996) besteht darin, aus der Zielbeschreibung mittels syntaktischer *Feature-Transformationen* eine Reihe sogenannter *Features* zu generieren, aus welchen dann in einem *Feature-Selektions*-Schritt die besten ausgewählt und zu einer Evaluationsfunktion kombiniert werden. Im Rahmen dieser Arbeit soll die wissenschaftliche Fragestellung der Übertragbarkeit dieses Verfahrens auf das General Game Playing untersucht werden. Anstelle der von Fawcett eingesetzten "Preference Pairs" zur Bestimmung der Feature-Gewichte soll Reinforcement Learning zum Einsatz kommen. Alle untersuchten Verfahren sollen zur experimentellen Unterlegung der Untersuchung in Prolog implementiert und in den General Game Player *Fluxplayer* integriert werden. Weiterhin sollen mögliche Verbesserungen vorgeschlagen, implementiert und evaluiert werden.  
Schwerpunkte:

- Übertragung von Fawcetts *Feature-Discovery*-Verfahren auf das General Game Playing und Implementation
- Kombination mit Reinforcement Learning
- Evaluation des Verfahrens sowie eigener Verbesserungen

Betreuer und  
verantwortlicher  
Hochschullehrer: Prof. Michael Thielscher  
Institut: Künstliche Intelligenz  
Lehrstuhl: Computational Logic  
Beginn am: 22.04.2008  
Einzureichen am: 22.10.2008

---

Verantwortlicher Hochschullehrer

## Literatur

Fawcett, T. E. (1996). Knowledge-based feature discovery for evaluation functions. *Computational Intelligence*, 12(1):42–64.



## Abstract

The goal of General Game Playing is to construct an autonomous agent that can effectively play games it has never encountered before. The agent is only provided with the game rules, without any information about how to win the game. Since no human intervention is allowed, the agent needs to deduce important concepts automatically from the game rules.

A central challenge is the automatic construction of an evaluation function that can estimate the winning chances for any position encountered during game-tree search. An evaluation function is a weighted combination of *features*: numerical functions that identify important aspects of a position.

This thesis presents a method to generate a set of features for General Game Playing, based on previous work on knowledge-based feature generation. Moreover, a method is developed that quickly selects features for inclusion in the evaluation function. The feature construction method is combined with TD( $\lambda$ ) reinforcement learning to produce a complete evaluation function.

It is shown that the method could successfully generate an evaluation function for many general games, and an empirical evaluation of its quality is presented.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Previous Work</b>	<b>3</b>
2.1	Feature Construction . . . . .	3
2.1.1	Multi-Layer Neural Networks . . . . .	3
2.1.2	ELF . . . . .	4
2.1.3	GLEM . . . . .	4
2.1.4	Zenith . . . . .	5
2.2	Adaptation to GGP . . . . .	10
2.2.1	Problems with Zenith’s Learning Algorithm . . . . .	11
2.2.2	Problems with Zenith’s Feature Selection . . . . .	11
2.2.3	Consequences for Feature Generation and Selection . . . . .	12
2.3	Game Description Language . . . . .	12
<b>3</b>	<b>Feature Generation</b>	<b>16</b>
3.1	Feature Formalism . . . . .	16
3.2	Feature Generation Algorithm . . . . .	17
3.3	Feature Transformations . . . . .	19
3.3.1	Abstraction Transformations . . . . .	19
3.3.2	Specialization Transformations . . . . .	22
3.3.3	Other Transformations . . . . .	23
3.3.4	Unimplemented Zenith Transformations . . . . .	27
3.4	Simplifier . . . . .	29
3.5	Trim-Variables . . . . .	30
3.6	Duplicate Removal . . . . .	32
3.7	Restricting the Transformations . . . . .	33
<b>4</b>	<b>Feature Selection</b>	<b>36</b>
4.1	Abstraction-Graph-Based Feature Selection . . . . .	36
4.2	Building the Abstraction Graph . . . . .	37
4.3	Assigning Abstraction Levels . . . . .	40
4.4	Final Feature Selection . . . . .	44
<b>5</b>	<b>Evaluation Function Learning</b>	<b>49</b>
5.1	Learning Framework . . . . .	49

5.2	Action Selection . . . . .	49
5.3	Evaluation Function Model . . . . .	52
5.4	TD Weight Update . . . . .	53
<b>6</b>	<b>Empirical Results</b>	<b>55</b>
6.1	Feature Generation and Selection . . . . .	55
6.1.1	Too Many Features . . . . .	55
6.1.2	Too Few Features . . . . .	56
6.1.3	Successful Feature Generation . . . . .	59
6.1.4	Cost of transformations . . . . .	59
6.2	Evaluation Function Learning . . . . .	64
6.2.1	Single-Player Games . . . . .	66
6.2.2	Turn-Taking Games . . . . .	68
6.2.3	Simultaneous Games . . . . .	69
<b>7</b>	<b>Conclusions</b>	<b>71</b>
7.1	Critical Assessment and Future Work . . . . .	71
7.1.1	Feature Generation . . . . .	71
7.1.2	Abstraction-Graph-Based Feature Selection . . . . .	72
7.1.3	Evaluation Function Learning . . . . .	74
7.2	Summary . . . . .	75
<b>A</b>	<b>Overview of the Games</b>	<b>77</b>
<b>B</b>	<b>Training graphs</b>	<b>81</b>
	<b>List of Figures</b>	<b>103</b>
	<b>List of Tables</b>	<b>105</b>
	<b>List of Algorithms</b>	<b>107</b>
	<b>List of Acronyms</b>	<b>109</b>
	<b>Bibliography</b>	<b>111</b>



# 1 Introduction

This thesis describes a method for automatic construction of features for utilization in a heuristic evaluation function. Based on previous work in the area of knowledge-based feature generation, its applicability to the domain of General Game Playing is evaluated.

General Game Playing (GGP) is the challenge to build an autonomous agent that can effectively play games that it has never seen before. Unlike classical game playing programs, which are designed to play a single game like Chess or Checkers, the properties of these games are not known to the programmer at design time. Instead, they have to be discovered by the agent itself at runtime. This demand for higher flexibility requires the use and integration of various techniques and makes GGP a grand Artificial Intelligence (AI) challenge.

In recent years, GGP has received an increasing amount of attention. To foster further research efforts in this area, the annual GGP competition (Genesereth, Love, and Pell, 2005) was established in 2005. Participating systems are pitted against each other on a variety of different types of games.

One of the central challenges in building a successful general game playing program lies in the fully automatic construction of an evaluation function. Such evaluation functions are essential for guiding search in many different fields of AI. In game playing, an evaluation function maps each game state to a real number that estimates the chances of winning the game from this position. A state is typically represented as a set of facts that uniquely define the state – in a board game, this could be the exact locations of all pieces on the board, in a card game it could be the cards held by each player, the number of accumulated points and so on. However, due to the large state space of most games, learning an evaluation function directly from this base-level representation of states is generally not feasible. For that reason, evaluation functions are usually defined as a combination of evaluation features.

Features are numerical functions of a state that capture important properties of the game. For a board game like chess, these could be the number of pieces of each type that a player has left, control of the center of the board or certain attacking relationships. Features provide a way to generalize over a set of states, thereby immensely simplifying the task of constructing an evaluation function. Using these features, the evaluation function can be constructed by selecting a functional form that combines the feature values into a single number, and adjusting the parameters (weights) of this functional form.

Following this approach, the process of creating an evaluation function can be

split into two distinct subtasks:

1. feature generation and selection, and
2. evaluation function learning (automatic adjustment of the weights of an evaluation function).

The second task has been studied extensively with great success. The most widely used methods are supervised learning from expert matches and different forms of reinforcement learning, including the TD( $\lambda$ ) algorithm.

Meanwhile, the first task (the problem of automated feature construction) is still open: “One of the key problems, that has already been mentioned in Samuel’s famous Checkers player (Samuel, 1959, 1967), namely the automated construction of useful features, remains still largely unsolved.” (Fürnkranz, 2007, p. 3).

Surprisingly few researchers have tried to tackle this problem – most notably Fawcett (1993), Utgoff and Precup (1998) and Buro (1999) – and there has not been much follow-up work on their approaches, despite the importance of the problem for automated construction of an evaluation function. One of the reasons may have been that game playing research in the past was focused on specific games, where features can be specified manually. But in the new context of GGP, automatic feature construction suddenly regains relevance.

Of the existing work on feature generation, Fawcett’s approach seems to be best suited for adoption to GGP, because it makes use of domain knowledge and requires no human intervention. Some of the previous work in GGP mention Fawcett’s work (Asgharbeygi, Stracuzzi, and Langley, 2006; Banerjee, Kuhlmann, and Stone, 2006), but none has actually applied his ideas to the domain of GGP yet.

The aim of this work is to develop, implement and evaluate a system consisting of

- a feature generation algorithm, based on Fawcett’s approach
- a feature selection algorithm, and
- an evaluation function learning algorithm, using TD( $\lambda$ ) learning

for the GGP framework. All algorithms are implemented in Prolog and integrated into the General Game Player “Fluxplayer” (Schiffel and Thielscher, 2007b).

The remainder of this work is organized as follows: Chapter 2 reviews the previous work on feature generation as well as the Game Description Language that is used to express games in GGP. Chapters 3, 4 and 5 describe the three phases of the implemented system: feature generation, feature selection and evaluation function learning. Chapter 6 evaluates the system’s performance on a variety of games. Chapter 7 concludes with a discussion of the achievements.

## 2 Previous Work

In this chapter, we will first give an overview of the previous work on feature construction. Next, the consequences from these previous approaches for the current thesis will be discussed. The chapter will conclude with an overview of the Game Description Language (GDL) (the language used to describe games in GGP), including the definitions that will be needed in the remainder of this thesis.

### 2.1 Feature Construction

The previous approaches to feature construction can be roughly separated into the following two categories: multi-layer neural networks and symbolic approaches. In the following two subsections, both will be examined.

#### 2.1.1 Multi-Layer Neural Networks

Multi-layer neural networks can be seen as feature constructing systems in the sense that the hidden layer of those networks combines the base-level representation of the game into intermediate concepts. These are combined into a single evaluation of a state by the output layer. The formation of those intermediate concepts is not done explicitly, but occurs during the learning phase of the network. Neural networks have been applied to a wide variety of games, including Backgammon (Tesauro, 1995), Othello (Leouski and Utgoff, 1996), Chess (Levinson and Weber, 2002; Baxter, Trigg, and Weaver, 1998), Go (Dahl, 2001) and Poker (Billings, Peña, Schaeffer, and Szafron, 2002).

However, approaches based on neural networks have two major drawbacks: firstly, the generated features cannot be interpreted directly by a human; and secondly, the structure and parameters of the neural network need to be carefully tailored to the specific application by hand, which makes direct application to GGP difficult. However, it should be noted that there has been some research on mapping symbolic domain theories into neural networks (Towell and Shavlik, 1994) that could solve the second problem.

The following subsections will deal with systems that create symbolic representations of features: ELF, GLEM, and Zenith.

### 2.1.2 ELF

The Evaluation Function Learner (ELF) algorithm (Utgoff and Precup, 1998) represents each feature as a boolean function of the algorithm’s boolean inputs<sup>1</sup>. Each feature is represented as a vector of the symbols  $\#$  and  $\theta$ ;  $\#$  (“don’t care”) means that the corresponding input can have any value,  $\theta$  (“false”) means that the input must be false. The feature only matches those states where all these conditions are met. Surprisingly, there is no way of expressing that an input must be true; however, the authors demonstrate that the feature formalism is still sufficient to express any evaluation function.

ELF can be embedded as a function approximator into any learning framework that provides  $\langle state, value \rangle$  pairs, such as supervised learning, comparison training, reinforcement learning and temporal-difference learning. During the training phase, ELF both updates the feature weights and adds new features as necessary.

Initially, the feature set only contains the most general feature (exclusively consisting of  $\#$ ’s); during training, the algorithm iteratively

1. identifies the feature that is least able to reduce its error
2. identifies the  $\#$  in the feature with the largest accumulated error, and
3. makes a copy of this feature where this  $\#$  has been replaced by a  $\theta$ .

ELF has been applied to Tic-Tac-Toe and the Eight-Puzzle, using supervised learning with the perfect game-theoretic state values (Tic-Tac-Toe) resp. the optimal distance to the goal (Eight-Puzzle) as training signal. ELF was also used to learn to play Checkers from self-play with limited success (Fürnkranz, 2001).

### 2.1.3 GLEM

The Generalized Linear Evaluation Model (GLEM) algorithm (Buro, 1999) constructs new features as conjunctions of boolean *atomic features*.

Contrary to ELF and Zenith, GLEM does not interweave feature generation and parameter tuning. Instead, all used features are generated beforehand. Since the number of possible conjunctions grows exponentially with the number of atomic features, the legal conjunctions are restricted to a user-defined set of *patterns* (in Buro’s experiments on Othello, these patterns only allowed conjunctions of atomic features of the same row, column, diagonal or corner regions of the board). To avoid over-specialization, all generated features are tested on a large set of example states, and those whose matching ratio lies below a certain threshold are excluded.

The final evaluation function combines the features linearly and applies a sigmoid squashing function – similar to that used in neural networks – to avoid saturation

---

<sup>1</sup>These inputs are called *state variables* by Utgoff and Precup and will be called *fluents* in this thesis.

effects. The weights are fitted using least-squares optimization on a labelled training set. One common way to generate this training set is from expert matches. Another way that is proposed by Buro is to generate the training set from the game-tree search itself: Using the fact that practically all Othello matches take between 58 and 60 plies, the game is partitioned into 15 stages, and a separate set of weights is calculated for each of these stages. First, an evaluation function for the final stage of the game (plies 57–60) is trained, generating the labelling by exhaustive search. Next, this evaluation function is used to label example states generated by game-tree search for the pre-final stage (ply 53–56). This labeled training set can be used to train an evaluation function for the pre-final stage and so on. During actual game-play, the correct evaluation function to use is determined by the ply number.

The atomic features could simply be all fluents of the game (e. g., the contents of each field of the board), allowing GLEM to generate – in principle – any possible evaluation function. This approach worked very well for Othello (in combination with the handcrafted patterns mentioned above). However, Buro points out that when using such a simple set of atomic features, important concepts of many other games, such as the “attacks” relationship in chess, have a very long description length and are unlikely to be discovered by the algorithm. Therefore, the set of atomic features should be carefully tailored to the specific application domain: “GLEM allows the program author to concentrate on the part of evaluation function construction, where humans excel: the discovery of fundamental positional features by *reasoning* about the game” (Buro, 1999, p. 143). Hence, GLEM should not be seen as a fully automatic feature construction algorithm, but rather as a system that eases the task of constructing features and reduces the amount of manual work required.

GLEM has been used to learn an evaluation function for Logistello, the best Othello-playing program of its time. The evaluation function created by GLEM greatly outperformed Logistello’s previous evaluation function that was based on purely handcrafted features.

#### 2.1.4 Zenith

Since this thesis will be mainly based on Fawcett’s work, his Zenith system (Fawcett, 1993, 1996; Fawcett and Utgoff, 1992) will be examined in more detail than the other two. Zenith has been applied to the game of Othello. It could successfully regenerate many known features from the literature, and even at least one novel feature.

Zenith is the only one of the three systems that uses analysis of a declarative domain theory (the game’s rules) for deriving the features. The language in which this domain theory is expressed is Prolog; arbitrary Prolog predicates are allowed, which makes Fawcett’s domain description language strictly more expressive than, for example, the more common STRIPS formalism.

Similar to GDL, the domain theory contains predicate declarations for precondi-

tions and effects of actions, terminal states and goal values<sup>2</sup>. Additionally, it contains information on the modes and determinacy<sup>3</sup> of all predicates, whether a predicate is state-specific or not, and the preimages of all state specific predicates (which greatly simplifies the implementation of a regression transformation later on). In contrast to GDL, the preconditions and effects of an action are specified explicitly for each action, making frame axioms unnecessary.

A Zenith feature is represented as a logical formula using terms from the domain theory, along with a variable list. The value of a feature in a state is defined as the number of unique bindings of variables in this list that satisfy the formula. Since this formalism is the basis of this thesis, it will be covered in detail in Chapter 3.

Starting from the goal concept, Zenith iteratively develops sets of features through a series of feature transformations (*feature generation* phase) and applies its learning algorithm to assign weights to each feature. The feedback from learning is used to guide the selection of used features (*feature selection* phase) before the next cycle starts. This process is depicted in Figure 2.1 on the facing page; the next two subsections will cover these two phases in greater detail.

## Feature Generation

Zenith keeps track of two separate sets of features: the *active set* and the *inactive set*. The active set contains all features that are currently used in the evaluation function; the inactive set holds a limited number of potentially valuable features that have not been selected by the feature selection phase (see below).

Zenith's feature transformations<sup>4</sup> are listed in Table 2.1 on page 8. During the feature generation phase, these transformations are applied to both active and inactive sets, following these rules:

1. decomposition transformations can be applied to all features;
2. abstraction and specialization transformations are only applied to *expensive* features (features whose computation time exceeds a certain threshold); and
3. goal regression is only performed on features that are both active and inexpensive.

---

<sup>2</sup>Curiously, the domain theory (Fawcett, 1993, Appendix A) does not contain any information on the initial state, but this is probably a detail due to Fawcett's separate implementation of the state representation.

<sup>3</sup>The *determinacy* of a predicate specifies what combinations of bound and unbound arguments in a call to this predicate will produce at most one solution.

<sup>4</sup>Since slightly modified versions of Fawcett's feature transformations are used in the current thesis, a more formal description of those transformations will follow in Section 3.3.

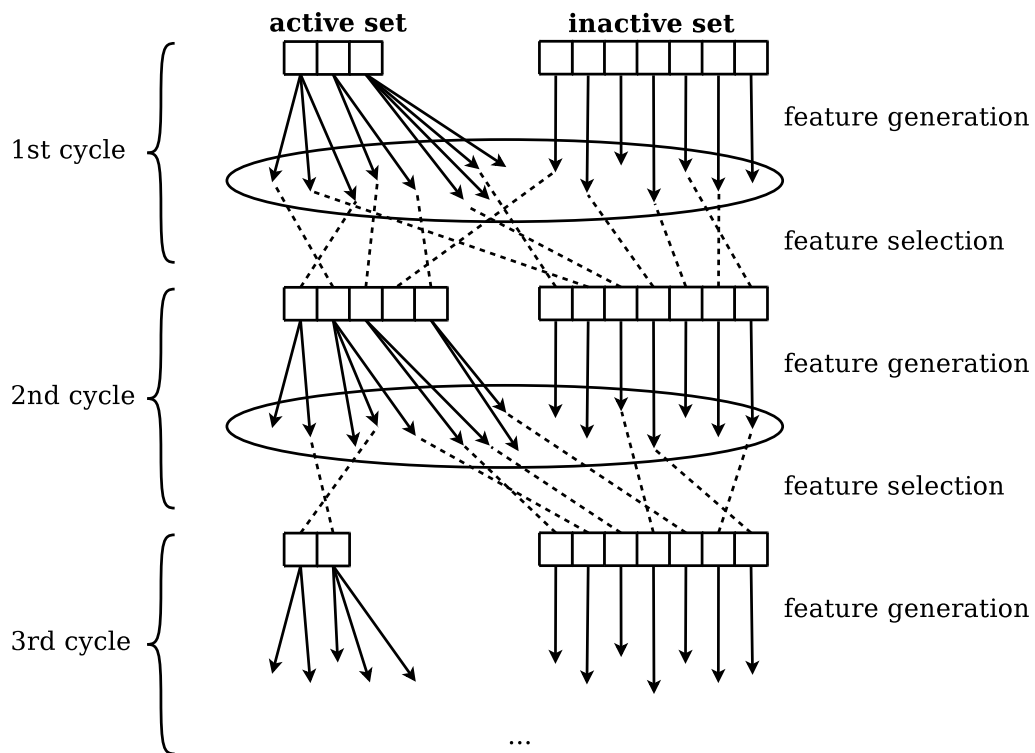


Figure 2.1: Feature generation and selection (source: Fawcett, 1993, modified)

Table 2.1: Transformations in Zenith (source: Fawcett, 1993)

<b>Class</b>	<b>Name</b>	<b>English description</b>
Decomposition	split-conjunction	Split conjunction into independent parts
	remove-negation	Replace $\neg P$ with $P$
	split-arith-comp	Split arithmetic comparison into constituents
	split-arith-calc	Split arithmetic calculation into constituents
Abstraction	remove-LC-term	Remove least constraining term of conjunction
	remove-variable	Remove a variable from the feature's variable list
Goal Regression	regress-formula	Regress a feature's formula through a domain operator
Specialization	remove-disjunct	Remove a feature's disjunct
	expand-to-base-case	Replace call to recursive predicate with base case
	variable-specialize	Find invariant variable values that satisfy a feature's formula

---



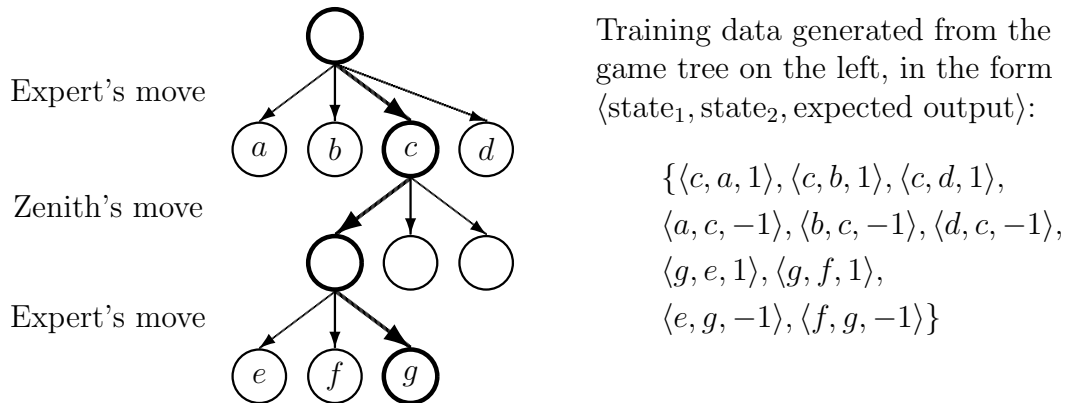


Figure 2.2: Preference Pair Learning

In order to restrict the number of generated features even more, only the first applicable feature transformation is performed in a cycle (however, a single transformation can generate several new features). If the feature is still present in a subsequent cycle, the next applicable feature transformation will be performed.

### Feature Selection

The purpose of the feature selection phase is to find a new set of active features such that

1. the features with the best predictive value are selected, and
2. the overall computation time of the evaluation function does not exceed a fixed threshold.

In order to attain a measure of the features' predictive values, Zenith constructs an evaluation function from all newly generated features, along with the old ones, and uses a learning algorithm to estimate their weights. The algorithm used for this is known as *preference pair learning* or *comparison training* (Tesauro, 1989). Instead of constructing a regular evaluation function, this algorithm learns a preference predicate that indicates which of two states is preferred over the other. The algorithm needs training data in the form of labeled preference pairs.

Zenith generates this training data by playing a single game, using the old evaluation function, against an expert opponent, in this case Wystan (an Othello playing program using 6-ply search and a hand-crafted evaluation function). From observing the expert's moves, more specifically which successor state was preferred over all other successor states, about 250 preference pairs per game can be inferred (this process is depicted in Figure 2.2). These are added to the training data collected during the previous cycles.

Using this training data, any supervised learning algorithm for training a function approximator can be employed. Fawcett performed experiments with both a linear threshold unit (perceptron) and a decision tree learning algorithm (named C4.5).

After the training algorithm has assigned weights to the features, a naïve method for selecting the best features would be to simply select those features with the greatest weights. However, this is not possible due to the statistical phenomenon of multicollinearity: many of the generated features' outputs are highly correlated, sometimes even identical, although the syntactic representation is different. While this does not reduce the predictive power of the whole evaluation function, it means that its weights become very volatile and may change erratically after removing even a single feature. Also, the relative importance of a feature can be either over- or underestimated.

The remedy chosen by Fawcett is called *sequential backward selection*: Instead of running the learning algorithm only once, it is run several times, starting with all (active and inactive) features. On each iteration, a different feature is left out.

Then, the classification accuracy of each of these reduced evaluation functions is tested on a validation set of preference pairs. This way, the *worst* feature (i. e., the feature whose removal has the smallest impact on the quality of the evaluation function) can be identified and cast out. Then, the procedure repeats until the overall computation time of the remaining features falls below the threshold. The remaining features form the new active set, all others the new inactive set. While this method is statistically sound and reliably eliminates the problem of multicollinearity, it involves repeating the whole learning process up to  $(n^2/2)$  times for  $n$  features in each iteration of the algorithm.

## 2.2 Adaptation to GGP

One key difference between the three approaches is that both ELF and GLEM make no use of a domain theory, while Zenith does. The GLEM algorithm compensates for this by allowing the user to manually specify atomic features, containing important concepts that are difficult to express directly as a conjunction of fluents. However, since such manual interference is not possible in GGP, and a domain description is available, Zenith was chosen as a basis for this thesis.

One of the premises of this work was that, contrary to Zenith, TD( $\lambda$ ) reinforcement learning should be used as a learning algorithm. The reason behind this decision and the consequences it has for the current thesis will be explained in the following subsections.

### 2.2.1 Problems with Zenith's Learning Algorithm

Fawcett notes that: “In most runs the classification accuracy of Zenith’s preference predicate rises almost monotonically; however, the corresponding effect on problem solving performance is much more erratic than would be expected.” (Fawcett, 1993, p. 115). He suggests several possible explanations for this phenomenon, including that some decisions are more important than others, and the learning algorithm may fail to identify critical turning points in a game. One of the remedies he suggests is the use of Temporal Difference (TD) learning.

Another possible explanation for this erratic behaviour is that the training set is enlarged by only a single match per cycle. In the reported experiments, Zenith was run for ten cycles, which means that Zenith’s whole game-play experience is based on merely ten matches at most, and considerably less for the first cycles.

### 2.2.2 Problems with Zenith's Feature Selection

The main problem with Zenith’s feature selection method is that the learning process has to be repeated  $O(cn^2)$  times, where  $n$  is the number of concurrent features and  $c$  the number of cycles (see Section 2.1.4). Fawcett notes that “feature selection remains one of the most expensive components of Zenith, and probably will remain so. Other selection methods were tested (Kittler, 1986; Kira and Rendell, 1992) but were either too expensive or failed to satisfy Zenith’s requirements for feature selection. Because of the criticality of feature selection, the search for an inexpensive but accurate selection method remains an important area of future work” (Fawcett, 1993, p. 114).

The fact that the learning process is repeated so often in Zenith has two important consequences. Firstly, a very fast learning algorithm and a small training set had to be chosen, even if it has certain disadvantages (see Section 2.2.1). Secondly, the number of features that are kept at any single time must be strictly limited in Zenith. The fact that Zenith uses an iterative approach makes the situation even worse, because learning has to start from scratch in each cycle. This explains the restrictions placed on feature generation and the limited number of features kept in the active and inactive feature sets; these restrictions make Zenith’s feature selection process a beam search. The greedy nature of a beam search makes it possible that important features are missed.

In Fawcett’s experiments with Othello, Zenith generated a total of about 200 features, of which only at most 30 were contained in the combined active and inactive feature sets, and 16 were finally selected after six days of computation<sup>5</sup>. To improve

---

<sup>5</sup>The absolute computation times mentioned cannot be compared directly, because the hardware Fawcett used in 1993 much slower than today’s hardware. The numbers are only reported to give the reader an impression of the time constraints.

the accuracy of the resulting evaluation function, it would be desirable if a larger portion of the feature space could be explored.

GLEM, on the other hand, generated and selected more than 100,000 features for Othello. Since the game was partitioned into 13 stages, each using its own evaluation function, more than a million weights had to be fitted using around 11 million training positions. This was accomplished in about 30 hours of computation time. Using this many features makes it less likely that important ones are missed; it shifts the task of selecting the important features from the feature selection to the parameter learning algorithm.

This huge difference in the feasible number of features can partly be explained by the fact that GLEM could use highly specialized Othello functions, while Zenith (due to its generality) had to interpret the domain theory. Also, GLEM's feature formalism (simple conjunctions of boolean values) allows to use very efficient methods for computing the matching features and for storing and updating the weights. Still, this number of weights could only be learned because the training algorithm only had to be run once. In fact, Buro states that "Taking into account the large number of features needed for an adequate evaluation in complex domains, and the resulting considerable effort for optimizing weights, it seems hopeless to combine feature construction and weight fitting" (Buro, 1999, p. 144).

### 2.2.3 Consequences for Feature Generation and Selection

In conclusion, the high complexity of sequential backward selection and the iterative approach forced Fawcett to severely restrict the number of generated features, and to use an inexpensive but inaccurate learning algorithm, combined with a tiny instance set, to keep the time complexity of Zenith manageable.

While TD learning could improve the actual game-play performance, it typically has to be run on at least hundreds of matches – it is more accurate, but also more expensive than preference pair learning. The consequence is that repeating the learning process thousands of times becomes infeasible.

These considerations lead to the decision to use a non-iterative approach to feature generation and feature selection in this thesis, combined with TD learning as the learning algorithm. In this non-iterative approach, feedback from the learning algorithm cannot be used to guide the selection of features, so a new method for feature selection has to be developed.

## 2.3 Game Description Language

The Game Description Language (GDL) (Genesereth et al., 2005) is the language used in GGP to communicate the rules of the game to each player. It is a variant of first order logic, enhanced by distinguished symbols for the conceptualization of

games. GDL is purely axiomatic, i. e. no algebra or arithmetics is included in the language. If a game requires some algebra or arithmetics, the relevant portions have to be axiomatized in the game description.

The class of games that can be expressed in GDL can be classified as *n-player* ( $n \geq 1$ ), *deterministic*, *perfect information* games with *simultaneous moves*. “Deterministic” excludes all games that contain any element of chance, while “perfect information” prohibits that any part of the game state is hidden from some players, as it is common in most card games. “Simultaneous moves” allows to describe games where all players move at once (like Roshambo), while still permitting to describe games with alternating moves (like Chess or Checkers) by restricting all players except one to a single “no-op” move. Also, GDL games are *finite* in several ways: The state space consists of finitely many states; there is a finite, fixed number of players; each player has finitely many possible actions in each game state; and the game has to be formulated such that it leads to a terminal state after a finite number of moves. Each terminal state has an associated goal value for each player, which needs not be zero-sum.

A game state is defined by a set of atomic properties, the *fluents*, that are represented as ground terms. One of these game states is designated as the initial state. The transitions are determined by the joint actions of all players. The game progresses until a terminal state is reached.

*Example 2.1.* Listing 2.1 on the next page shows the GDL game description<sup>6</sup> of the game Tic-Tac-Toe<sup>7</sup>.

The `role` keyword (lines 1–2) declares the argument, `xplayer` resp. `oplayer`, to be a player in the game.

The initial state of the game is described by the keyword `init` (lines 4–9). Initially, all cells are blank (`b`) and `xplayer` is first to move.

The keyword `next` (lines 11–16) defines the effects of the players’ actions. For example, line 11 declares that, after `xplayer` has executed action `mark(M, N)`, the fluent `cell(M, N, x)` will hold in the resulting state, meaning that `cell(M, N)` is marked with an `x`. The reserved keyword `does` can be used to access the actions executed by the players, while `true` refers to all fluents that are true in the current state. GDL also requires the game designer to state the non-effects of actions by specifying frame axioms, as can be seen on line 14: A cell that is not blank will still have its current mark in the resulting state, regardless of the players’ actions.

The keyword `legal` (lines 19–24) defines what actions are possible for each player in the current state; the game designer has to ensure that each player always has at least one legal action available. All GDL games have simultaneous moves; games with non-simultaneous moves, like Tic-Tac-Toe, can be expressed by introducing a

<sup>6</sup>We use Prolog notation with variables denoted by uppercase letters. All reserved GDL keywords are shown in bold.

<sup>7</sup>A short description of the games used for this thesis can be found in Appendix A.

**Listing 2.1** Some GDL rules of the game Tic-Tac-Toe

---

```
1 role(xplayer).
2 role(oplayer).
3
4 init(cell(1, 1, b)).
5 init(cell(1, 2, b)).
6 init(cell(1, 3, b)).
7 ...
8 init(cell(3, 3, b)).
9 init(control(xplayer)).
10
11 next(cell(M, N, x)) :-
12   does(xplayer, mark(M, N)).
13
14 next(cell(M, N, Mark)) :-
15   true(cell(M, N, Mark)),
16   Mark \= b.
17 ...
18
19 legal(Role, mark(M, N)) :-
20   true(cell(M, N, b)),
21   true(control(Role)).
22
23 legal(xplayer, noop) :-
24   true(control(oplayer)).
25 ...
26
27 goal(xplayer, 100) :-
28   line(x).
29 ...
30
31 terminal :-
32   (line(x); line(o); not open).
33
34 open :-
35   true(cell(M, N, b)).
36
37 line(Mark) :-
38   (row(M, Mark);
39    column(N, Mark);
40    diagonal(Mark)).
41
42 row(M, Mark) :-
43   true(cell(M, 1, Mark)),
44   true(cell(M, 2, Mark)),
45   true(cell(M, 3, Mark)).
46 ...
```

---

fluent that tracks which role is next to move (here called **control**) and only allowing non-effect moves (here called **noop**) for the other players, as can be seen on line 23.

The **goal** predicate (lines 27–28) assigns a number between 0 (loss) and 100 (win) to each role in a terminal state. The game is over when a state is reached where the **terminal** predicate (lines 31–32) holds.

Lines 34–45 show some of the auxiliary predicates defined in Tic-Tac-Toe.

Since GDL is under constant development, we will now formally define what will be considered a legal GDL formula in this thesis. This vocabulary forms the basis for definitions later in this thesis.

**Definition 2.1** (Term). A *term* is a variable or a function symbol applied to terms as arguments (a *constant* is a function symbol with no argument).

**Definition 2.2** (Atom). An *atom* is a predicate symbol applied to terms as arguments.

**Definition 2.3** (Formula). A *formula* is defined inductively as follows:

- If  $A$  is an atom, then  $A$  is a formula;
- if  $T_1$  and  $T_2$  are terms, then both  $T_1 = T_2$  and  $T_1 \neq T_2$  are formulae;

- if  $F$  is a formula, then  $\neg F$  is a formula;
- if  $F$  and  $G$  are formulæ, then both  $F \wedge G$  and  $F \vee G$  are formulæ;
- nothing else is a formula.

The current GDL specification puts additional restrictions on valid GDL formulæ:

1. equality ( $=$ ) is not allowed;
2. disjunctions are not allowed; and
3. negations are only allowed directly in front of atoms.

The first restriction was omitted here to make the implementation of several transformations more straightforward, but they could easily have been implemented without equality. The second and third restriction were omitted in order to maintain compatibility with older GDL games that use rules of that kind.

**Definition 2.4** (Clause). A **clause** is an implication “ $H \Leftarrow B$ ” or “ $H \Leftarrow$ ”, where **head**  $H$  is an atom and **body**  $B$  is a formula.

**Definition 2.5** (Predicate). A **predicate** is a collection of all clauses of a game description whose heads all have the same predicate symbol and arity.

**Definition 2.6** (Fluent). A **fluent** is a ground term whose function symbol occurs as an argument in the game description’s *init* or *next* predicates.

Fluents represent the atomic properties of a game. For example, `cell(1,3,x)` is a fluent that occurs in Tic-Tac-Toe.

**Definition 2.7** (State). A **state** is a set of fluents.

An example for a state that can be reached in Tic-Tac-Toe is

$$\{\text{cell}(1,1,\text{x}), \text{cell}(1,2,\text{x}), \text{cell}(1,3,\text{o}), \text{cell}(2,1,\text{o}), \text{cell}(2,2,\text{b}), \\ \text{cell}(2,3,\text{b}), \text{cell}(3,1,\text{b}), \text{cell}(3,2,\text{b}), \text{cell}(3,3,\text{o}), \text{control}(\text{xplayer})\} \quad .$$

The state is used in GDL in the following way: In GDL, there is always assumed to be a “current state” with respect to which any GDL formula is evaluated. This state does not explicitly occur in the formula; instead, the special GDL predicate `true` evaluates to *true* for all elements of the current state and to *false* otherwise.

## 3 Feature Generation

In this chapter, we will first formally define the feature formalism that constitutes the basis of this work. Then, the feature generation algorithm will be described.

### 3.1 Feature Formalism

In Zenith’s feature formalism, which will also be used in this work, a feature is represented using a formula and a variable list. Intuitively, the evaluation of a feature is the number of possible bindings for the variables in the variable list. The following definitions capture these notions more formally.

**Definition 3.1** (Feature Formula). *Let  $\Delta$  be a GDL game description<sup>1</sup>, and let  $G$  be the dependency graph for  $\Delta$ . Then,  $F$  is called a **feature formula**, if each atom with predicate symbol  $P$  that occurs in  $F$  satisfies the following conditions:*

1.  $P$  is neither *role*, *init*, nor *next*, and
2. in  $G$ , there is no path between  $P$  and *does*.

Basically, any GDL formula that would be valid as the body of a **goal** or **terminal** rule is a valid feature formula.

**Definition 3.2** (Variable List). *A **variable list** for feature formula  $F$  is a list (or vector) whose elements are a subset of all free variables that occur in  $F$ , including the full and the empty list.*

**Definition 3.3** (Feature). *A **feature** is a pair  $\langle F, \vec{v} \rangle$ , where*

- $F$  is a valid feature formula, and
- $\vec{v}$  is a valid variable list for  $F$ .

The evaluation of a feature is a function that maps each state  $z$  to a number  $n \in \mathbb{N}$ . This feature value  $n$  is defined to be the number of distinct bindings of variables in the feature’s variable list that satisfy the feature’s formula in  $z$ . More formally:

---

<sup>1</sup>For a full formal definition of the terms *game description* and *dependency graph*, see the official GDL specification (Love, Hinrichs, Haley, Schkufza, and Genesereth, 2008). All restrictions placed on the dependency graph also apply here.



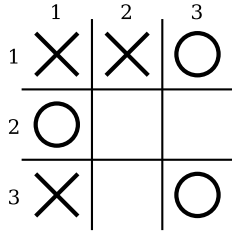
	Variable list	Variable bindings satisfying formula	Evaluation
	$[\ ]$	$\{[\ ]\}$	1
	$[N]$	$\{[1], [3]\}$	2
	$[M_1]$	$\{[1], [2]\}$	2
	$[M_1, M_2]$	$\{[1, 3], [2, 3]\}$	2
	$[M_1, M_2, N]$	$\{[1, 3, 1], [2, 3, 1], [1, 3, 3]\}$	3
	$\dots$	$\dots$	$\dots$

Figure 3.1: Evaluations of five features using different variable lists and the feature formula  $\text{true}(\text{cell}(M_1, N, x)) \wedge \text{true}(\text{cell}(M_2, N, o))$

**Definition 3.4** (Evaluation). *Let  $\varphi = \langle F, \vec{v} \rangle$  be a feature, and let  $\vec{w}$  be the vector of all free variables in  $F$  but not in  $\vec{v}$ . Then, the set  $S$  is defined as*

$$S := \{ \vec{v} \mid \exists \vec{w}. F(\vec{w}, \vec{v}) \} \quad (\text{relative to state } z),$$

and the **evaluation** of  $\varphi$  in  $z$  is defined as  $\text{eval}_\varphi(z) := |S|$ .

This definition implies that for a feature with  $\vec{v} = [\ ]$ ,  $S = \{[\ ]\}$  and therefore  $\text{eval}_\varphi(z) = 1$  if the formula  $F$  can be satisfied in  $z$ , and  $\text{eval}_\varphi(z) = 0$  otherwise.

As an abbreviation, if  $\text{eval}_\varphi(z) > 0$  for a feature  $\varphi$  and a state  $z$ , we will say that “ $\varphi$  **matches** in  $z$ ”.

This feature evaluation process is demonstrated in Figure 3.1. The evaluations of five different features are shown, all using the same feature formula, but a different variable list.

## 3.2 Feature Generation Algorithm

The purpose of the feature generation phase is to generate a large set of features from the game description. From this set, the feature selection algorithm (Chapter 4) will select a smaller set of features for inclusion in the evaluation function. Eventually, the evaluation function learning algorithm (Chapter 5) will run training matches to learn the weights of the evaluation function.

An overview of the FEATUREGENERATION algorithm is given in Figure 3.2 on the next page. Its general structure is to iteratively generate a series of feature sets until no more features can be generated.

The initial set of features is created from the `goal` and `terminal` predicates of the current game description. Each clause of these predicates becomes the formula of a new feature. The initial features start with a full variable list.

Starting with this initial feature set, the algorithm passes each feature in the current set to several *feature transformations*, which generate several new features

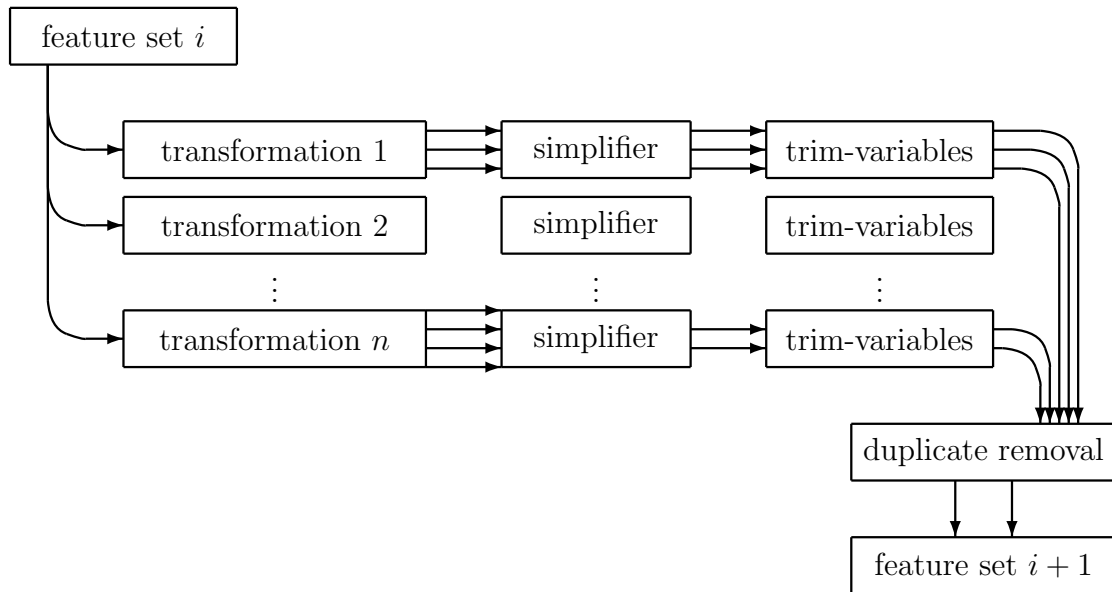


Figure 3.2: Overview of the feature generation process

from it. Not all feature transformations are applicable to all features, so there may be transformations which generate no new features, for example transformation 2 in the figure.

Each generated feature is passed through a *simplifier*, which tries to apply syntactic simplifications to a feature’s formula. The simplifier may determine that a formula is unsatisfiable, in which case the feature is rejected; otherwise, it is passed on through an optimization called *trim-variables*, which will be explained in Section 3.5 and can be ignored for now.

Afterwards, all newly generated features are passed to the *duplicate removal* procedure, which checks whether a feature has been generated before. If that is the case, the duplicate feature will be dropped. All remaining features form the next feature set. The algorithm is repeated until no more features can be generated. At this point, all generated feature sets are unified and returned.

Each part of the algorithm will be presented in the following subsections: the feature transformations (Section 3.3), the simplifier (Section 3.4), the trim-variables optimization (Section 3.5) and duplicate removal (Section 3.6). Section 3.7 will discuss additional restrictions to the transformations.

## 3.3 Feature Transformations

The feature transformations implemented for this work are mostly identical to those used in Zenith. Where the difference in domain languages made it necessary to change a transformation, it will be pointed out in the respective subsection.

In this thesis, the transformations are grouped into abstraction, specialization and other transformations. Abstraction transformations have the property that the resulting feature will match all states that the original feature matched, and possibly some more; the opposite is true for specialization transformations. All features that fit in neither of these two groups have been placed into the third group (“other transformations”). The notions of abstraction and specialization will be formally defined in Chapter 4. Strictly speaking, the counterpart of “special” would be “general” and that of “abstract” would be “concrete”. Nevertheless, the terms “abstraction” and “specialization” will be used to keep the terminology consistent with Fawcett and other related work such as Prieditis (1993).

Whenever a transformation is applied, the newly generated feature inherits the variable list from its parent (exceptions will be mentioned in the description of the respective transformation). Variables that are not present any more in the new feature’s formula are removed, variables that have been newly introduced are added to the list.

### 3.3.1 Abstraction Transformations

#### Split-Indep-Conjunctions

This transformation splits a feature with a conjunctive formula into independent parts, based on their common use of variables.

First, the original formula is split into its conjuncts. Each conjunct is also a formula: an atom, a negated formula, a disjunction, an equality or inequality. Next, the variables used in each conjunct are determined; whenever two conjuncts share at least one variable, they are placed into the same set. Finally, a new feature is created from each of these sets whose formula is a conjunction of the set elements.

For example, the feature

$$\langle (p(W, X), q(X, Y), r(Y), s(Z), t(u)), [X, Z]) \rangle$$

would be split into the following parts:

1.  $\langle (p(W, X), q(X, Y), r(Y)), [X] \rangle$ , based on the variables  $X$  and  $Y$ ;
2.  $\langle s(Z), [Z] \rangle$  – the variable  $Z$  appears in no other conjunct; and
3.  $\langle t(u), [] \rangle$  – this term contains no variables at all.

The justification for this transformation is that the satisfiability of each subformula can be established independently of the other formulæ, and splitting them into separate features both reduces the feature’s complexity and allows the learning algorithm to assign independent weights to each part.

### Remove-Conjunct

This transformation removes a single conjunct from a feature’s formula. Fawcett called this transformation **remove-LC-term** (short for “remove least constraining term”; “term” is used here in the Prolog sense of the word). He distinguished three classes of “criticality” of terms:

1. state-dependent terms that can be achieved by one of the actions of the player controlled by Zenith;
2. state-dependent terms that cannot be achieved by the Zenith player; and
3. state-independent terms.

Zenith’s **remove-LC-term** transformation only removed one of the least critical terms of a formula. Due to Fawcett’s implicit handling of states, he had to require that the state-dependency of all predicates had to be given explicitly in the game description, which allowed him to distinguish criticalities 1 and 2 from criticality 3. To distinguish between criticalities 1 and 2, Zenith calculated all regressions of the formula and checked whether the term was still present in all of the formula’s pre-images; if it was, the term was assigned a criticality of 2, otherwise 1.

In this implementation, the decision was made not to distinguish between criticalities 1 and 2. The reason is that in GGP there is no clear distinction between our player’s actions and the opponent’s actions: all players move simultaneously. Games with non-simultaneous moves are simulated by only allowing a “no-op” move for all players except the one in control. While deeper analysis of the game could reveal these no-op moves, there are still games (for instance, Merrills) where the same player can stay in control for several moves. Thus, using a single step of goal regression to determine whether a given fluent can be achieved by a certain role was deemed too unreliable. This decision means that **remove-conjunct** is more general than Zenith’s **remove-LC-term**, since it produces all features that **remove-LC-term** did, plus some additional ones.

The current **remove-conjunct** transformation still distinguishes between state-dependent and state-independent formulæ. In GDL games, it is not necessary to state the state-dependency of a predicate explicitly; instead, the following definitions were used that can be checked statically:

**Definition 3.5** (State-Dependent Predicate & Formula). *Let  $G$  be the dependency graph for a game description  $\Delta$ . Then, a predicate  $P$  is called a **state-dependent predicate** if there is a directed path between  $P$  and **true**, or if  $P$  equals **true**.*

A formula  $F$  is called a **state-dependent formula** if it contains an atom whose predicate symbol belongs to a state-dependent predicate.

Using this definition, `remove-conjunct` checks for each conjunct of the original feature’s formula whether it is state-dependent or not and only removes a conjunct if it is state-dependent. For each such conjunct, a new feature is generated that has the original feature’s formula except this conjunct.

The reason why state-independent formulæ are not removed is that they provide a “skeleton” of constraints on the feature’s variables that must be met in any state, and removing them almost always leads to bad features. The presence of fluents, and therefore the satisfiability of state-dependent formulæ, on the other hand, may change over time. For example, consider the following feature from the game Connect-Four:

$$\langle (\text{true}(\text{cell}(X1, Y, w)) \wedge \text{succ}(X1, X2) \\ \wedge \text{true}(\text{cell}(X2, Y, w)) \wedge \text{succ}(X2, X3) \\ \wedge \text{true}(\text{cell}(X3, Y, w)) \wedge \text{succ}(X3, X4) \\ \wedge \text{true}(\text{cell}(X4, Y, w))), [X1, X2, X3, X4] \rangle .$$

While removing any of the `true` atoms will produce a valuable feature, removing either of the `succ` atoms – which state that the second argument must be the number succeeding the first – will not.

Fawcett sees the purpose of this transformation mainly in the reduction of computation cost, and states that it produces features which provide less information about a state; while this is true, it can also be argued that `remove-conjunct` can sometimes produce features which generalize better and match more states, since irrelevant details are removed. In the extreme case, one of the conjuncts of a formula depends on a fluent which is only true in terminal states but has little significance with respect to the goal values. In such a case, without removing this conjunct, the feature could provide no information at all about non-terminal states.

### Remove-Variable

This transformation removes one variable from the feature’s variable list, while leaving the formula unchanged. This produces less expensive features and also allows the creation of features that concentrate on counting one aspect of a feature, thereby allowing the learning algorithm to assign separate weights to separate aspects. For example, in Chess there may be a feature that counts the number of white pawns attacking the number of black pawns; it might be useful to know the number of attacking white pawns, as well as the number of black pawns under attack, instead of just knowing the product of the two.

Since a feature produced by `remove-variable` provides less information about a state than the originating feature, `remove-variable` is also – for the time being –

counted as an abstraction information, although the resulting feature matches the same states as the original one.

### 3.3.2 Specialization Transformations

#### Split-Disjunction

Zenith contained a transformation called `remove-disjunct`, which simply removed a disjunct occurring in a feature’s formula. In this thesis, however, the choice was made to use a new transformation, called `split-disjunction`, instead. It works only on features that have a disjunctive formula, and splits this formula completely, generating one new feature from each disjunct. This is possible because, in contrast to a conjunction, the satisfiability of formulæ inside a disjunction can be established independently. The reduction in complexity – as opposed to generating all possible subformulæ, as `remove-disjunct` would do – is equal to that of `split-indep-conjunctions`.

#### Expand-to-Base-Case

This transformation only works on conjunctive formulæ which contain an atom whose predicate symbol belongs to a recursive predicate, replacing this atom by one of the predicate’s base-case clauses. To determine whether a predicate is recursive and what the base cases are, the following definition is used:

**Definition 3.6** (Recursive Predicate, Base Case). *Let  $\Delta$  be a GDL game description, and let  $G$  be the dependency graph for  $\Delta$ . Then, predicate  $P$  is called **recursive** iff there is a loop in  $G$  involving  $P$ .*

*A clause  $C$  of a recursive predicate  $P$  is called **base case** iff none of the atoms occurring in  $C$ ’s body is contained in a loop involving  $P$ .*

Like `remove-conjunct`, `expand-to-base-case` only operates on state-dependent atoms. It generates new features by unifying one of these atoms with the head of the chosen base-case clause and replacing the atom with the body of the unified clause.

#### Expand-Predicate

Interestingly, Zenith contained no transformation that handled non-recursive auxiliary predicates. The reason was probably that the Othello domain specification used by Fawcett contained no non-recursive auxiliary predicates, except state-independent ones that shouldn’t be expanded (like `neighbor`).

For this reason, the additional transformation `expand-predicate` is introduced. It operates similar to `expand-to-base-case`, except that all clauses of a non-recursive state-dependent predicate are eligible for expansion.

The justification for `expand-predicate` is that it allows other transformations to access the predicate's definition.

### 3.3.3 Other Transformations

#### Remove-Negation

`Remove-negation` only operates on features whose whole formula is a negated sub-formula. It creates a new feature by removing the negation. Since Prolog uses the negation-as-failure semantic, it is not possible to count the solutions of a negated formula; thus, features with a negated formula are always binary. By removing the negation, it becomes possible to count the number of solutions.

#### Regress-Formula

The transformation `regress-formula` performs goal regression on a feature's formula to create its pre-images. This is a very important transformation for the feature generation process, since it's the only one that takes the rules for legality of moves and successor states into account; without it, all features would only be grounded on the `goal` and `terminal` axioms.

Most STRIPS-style planners use goal regression extensively; however, regression in STRIPS is easy to implement because STRIPS domain theories have to explicitly supply a set of *preconditions*, a *deletelist* and an *addlist* for each domain operator, and because only literals are allowed for these lists.

However, many complex games like Othello cannot be expressed in such a constrained formalism, while they can in GDL or the Prolog domain theories used in Zenith. Unfortunately, this means that regression is more difficult to implement for these languages. For this reason, Fawcett required that the domain theory explicitly specifies the pre-images of any operator with respect to any state-dependent predicate in the domain specification. Since this explicit regression information is not available in GDL games either, an automatic procedure for computing pre-images is needed.

An initial attempt to implement this transformation used the regression operator described by Kostenko (2007). However, since this regression operator was intended for the creation of end-game databases, it attempts to produce a state description that is as concrete as possible, i. e., it expands all state-dependent predicates and creates all possible instantiations of variables in the formula. However, this is not desirable for feature generation, because a) the number of produced features is extremely large, and b) each of the produced features matches only a tiny number of states and therefore generalizes badly.

Thus, a different algorithm was devised. The following definition lies at its core:

**Definition 3.7** (Potential Preimage). *Let  $M = \{(R_1, A_1), \dots, (R_n, A_n)\}$  be a joint move, and let there be a game rule  $\text{next}(F) \Leftarrow B$  such that  $B$  does not imply  $\text{true}(F)$  and  $B$  is compatible with*

$$\exists R_1, \dots, R_n, A_1, \dots, A_n. (\text{does}(R_1, A_1) \wedge \dots \wedge \text{does}(R_n, A_n)) \quad .$$

Then, the formula

$$\text{legal}(R_1, A_1) \wedge \dots \wedge \text{legal}(R_n, A_n) \wedge B'$$

is called a **potential preimage** of fluent  $F$  for joint move  $M$ , where  $B'$  has been obtained from  $B$  by replacing every occurrence of  $\text{does}\langle R', A' \rangle$  by  $\langle R', A' \rangle = \langle R_1, A_1 \rangle \vee \dots \vee \langle R', A' \rangle = \langle R_n, A_n \rangle$ .

In the definition above, compatibility means logical consistency under the condition that each player can do only one action at a time. The requirement that  $B$  must not imply  $\text{true}(F)$  ensures that only non-frame axioms are counted. The intended meaning of “potential preimage” is as follows: Let  $Z_1$  be a state in which a fluent  $F$  does not hold, and let  $Z_2$  be the successor state of  $Z_1$  that was reached via joint move  $M$ , and in which  $F$  holds. In that case, at least one of the potential preimages of  $F$  for  $M$  holds in  $Z_1$ .

Using this definition, all preimages in a given game description can be calculated as follows:

1. Calculate all potential joint moves. This is done by collecting all actions for each role from the **legal** axioms, and generating the cross product of all these sets.
2. Collect all fluent symbols and their arity that appear in the **next** and **init** axioms of the game description.
3. Using Definition 3.7, calculate all potential preimages for all joint moves and all fluents.

Definition 3.7 also entails formulæ that are unsatisfiable in all reachable states. For example, if  $M$  contains the elements  $\langle R', A' \rangle$  and  $\langle R'', A'' \rangle$ , then there may be no reachable state where both  $A'$  is a legal move for role  $R'$  and  $A''$  is a legal move for role  $R''$ . Eliminating the preimages for such spurious moves would in general require to traverse the entire state space. But although solving the general case is intractable, an important special case can be handled: games with non-simultaneous moves. These are usually specified using “no-op” actions. Recognizing them would enable the algorithm to reduce the number of generated preimages immensely: in step 1 of the algorithm, only those joint moves could be included in which at most one of the roles executes an action different from the no-op action.



**Definition 3.8** (No-op action). *An action  $N$  is called a **no-op action**, iff*

- *it is a constant (i. e., has arity 0), and*
- *in every reachable state of the game, any legal joint move  $\{(R_1, A_1), (R_2, A_2), \dots, (R_n, A_n)\}$  contains at most one  $A_i \neq N$ .*

The requirement that the no-op action must be a constant was introduced only to improve the robustness of the no-op detection function. There are a few games in which the no-op action is represented by a function with arity  $\geq 1$ , for example `move(nowhere)`; these cases are not covered by the current definition.

Since it is very hard to prove automatically whether a given action is a no-op action or not, the current implementation uses simulation on a large number of games to determine whether a game contains a no-op action. This is only an approximation, but since this information is only used to restrict the number of features generated, the advantage of generating exponentially less features outweighs the disadvantage of potentially missing a preimage.

There is another optimization which reduces the number of generated regressions even further. GDL games often contain fluents which change at every step. We will call these always-changing fluents:

**Definition 3.9** (Always-Changing Fluent (ACF)). *A fluent  $F$  is called an **always-changing fluent**, if for all reachable states  $Z$  the following holds:*

$$Z' \text{ is a successor state of } Z \implies \neg(F \in Z \wedge F \in Z')$$

Typical examples of ACFs are step counters (to ensure that a game terminates) or control fluents that store which player's turn it is in non-simultaneous games. These shouldn't be regressed, because they change on every step and are an extreme case of an unstable fluent. Achieving these fluents will usually not improve game-play performance.

For this reason, all ACFs are excluded in step 2 in the algorithm above. Testing whether a fluent is an ACF is done by simulation, similar to the no-op actions.

An example of the generated preimages for the game of Pacman is shown in Listing 3.1 on the following page.

So far, we have only calculated the preimages of fluents, not predicates. More specifically, only state-dependent predicates would have to be regressed, since state-independent predicates are invariant to regression.

One way to calculate the preimage of a state-dependent predicate would be as follows: first expand the predicate, i. e., replace the predicate call by its definition, and then replace all predicate calls in the expanded formula, until only state-independent predicate calls and fluents are left. However, this would increase the formula length considerably, to the point where it may become intractable for very complex predicates. Even for those features where this complete expansion is feasible, the resulting

**Listing 3.1** The fluent preimages calculated for the game of Pacman

---

```

1 preimage(location(P,X2,Y2),
2   (legal(pacman,move(D1))),
3   legal(blinky,move(D2))),
4   legal(inky,move(D3))),
5   movable(P),
6   true(location(P,X1,Y1)),
7   ((P,move(D)) = (pacman,
8     move(D1))
9   ;(P,move(D)) = (blinky,
10    move(D2))
11   ;(P,move(D)) = (inky,
12    move(D3))),
13   nextcell(D,X1,Y1,X2,Y2))).
14 preimage(collected(N2),
15   (legal(pacman,move(D1))),
16   legal(blinky,move(_)),
17   legal(inky,move(_)),
18   true(location(pacman,X1,Y1)),
19   nextcell(D1,X1,Y1,X2,Y2),
20   true(location(pellet,X2,Y2)),
21   true(collected(N1)),
22   succ(N1,N2))).

```

---

long formula would not be very desirable, since the total number of features that are generated from a given feature rises exponentially with its formula length.

Alternatively, predicates could be regressed by creating a new predicate, for example called `preimage-of-p` for each predicate `p`. The definition of `preimage-of-p` would be identical to that of `p`, except that all fluents occurring in this definition would be substituted by their preimage, and all predicate calls `q` would be substituted by `preimage-of-q`. One clause would have to be added for each clause of the original predicate and each possible joint move. The drawback to this approach would be that lots of new predicates would have to be added to the game description, thereby making the game description much more complex. Eventually, the number of generated features would also increase dramatically: the regressed formula would bear almost no resemblance to the original formula, thereby opening up a whole new hierarchy of features generated by the other transformations. Another drawback of this approach is that either a regressed formula could not be regressed again, or one would have to introduce predicates like `preimage-of-preimage-of-p`.

Even if predicates were regressed that way, there would still be the problem of recursive predicates. Due to the properties of GDL, recursive predicates are guaranteed to terminate at some point; however, completely unrolling a recursive predicate is practically not feasible in most cases.

Therefore, the implementation of `regress-formula` used in this work only regresses a single fluent in the feature's formula, replacing the `true` statement containing the fluent by that fluent's preimage. Contrary to the two approaches discussed above, this only makes a small, local change to the feature's formula and does not cause an explosion in the number of generated features.

The disadvantage of the chosen approach is that now different parts of the formula refer to different states. In order for the new formula to be a true partial preimage of the old formula, one would have to add the atom `state_update(Z1, M, Z2)` to the formula. Assume that the predicate `state_update` is defined elsewhere and is

true iff  $Z2$  is the successor state of  $Z1$  for joint move  $M$ . The variable  $M$  would have to be instantiated with the joint move through which the fluent is regressed. Then the preimage could refer to the state  $Z1$ , whereas the non-regressed part of the formula would have to refer to  $Z2$ .

Unfortunately, using the predicate `state_update` in feature formulæ (perhaps even repeatedly in case of repeated regressions) would amount to a tree search; this would make the generated features prohibitively expensive and defeat the purpose of an evaluation function. The predicates used in a feature should refer exclusively to the current state.

Therefore, the decision was made to let the regressed part of the formula refer to the same state as the non-regressed part. Of course, this can lead to the generation of spurious features, like the following from Pacman:

$$\langle (\text{true}(\text{location}(\text{pacman}, X1, Y1)) \wedge \text{nextcell}(\text{Dir1}, X1, Y1, X2, Y2) \\ \wedge \text{legal}(\text{pacman}, \text{move}(\text{Dir2})) \wedge \text{nextcell}(\text{Dir2}, X2, Y2, X3, Y3) \\ \wedge \text{true}(\text{location}(\text{blinky}, X3, Y3))), [\text{Dir2}] \rangle .$$

The predicate call `legal(pacman, move(Dir2))` actually refers to the current state, where `pacman` is at cell  $(X1, Y1)$ , whereas the predicate call `nextcell` in which `Dir2` occurs refers to a state where `pacman` is at cell  $(X2, Y2)$ .

However, similar spurious features can be created by other transformations as well, for example if `remove-conjunct` drops a critical fluent. Recognizing and eliminating worthless features is the responsibility of the later stages of the algorithm. If this cannot be done by the feature selection step, this is accomplished by the learning algorithm, which will assign a near-zero weight to such features. Also note that a correct preimage can always be generated by the algorithm<sup>2</sup> by first expanding all state-dependent predicates (using `expand-predicate`) and then regressing every single fluent once through the same joint move.

### 3.3.4 Unimplemented Zenith Transformations

#### Split-Arith-Comp and Split-Arith-Calc

Zenith additionally contained two transformations called `split-arith-comp` and `split-arith-calc`. They work on formulæ which contain arithmetic comparisons and calculations. For example, the feature

$$\langle (p(A) \wedge q(A) \wedge r(B) \wedge s(C, D) \wedge \text{is}(E, ((A + B) \cdot C/D)) \wedge t(E)), [] \rangle$$

would be split into the following three features:

<sup>2</sup>This assumes that the complexity of the formula is low enough that this sequence of transformations is allowed by the transformation restrictions, see Section 3.7.

1.  $\langle\langle p(A) \wedge q(A), [v(A)] \rangle\rangle$
2.  $\langle\langle r(B), [v(B)] \rangle\rangle$
3.  $\langle\langle s(C, D), [v(C), v(D)] \rangle\rangle$

The predicate `is` is detected as a special predicate by the transformations indicating that the variables `A`, `B` and `C` should be treated as numbers. To be able to do so, Fawcett altered the definition of a feature’s evaluation: If the variable list contains  $v(X)$  instead of just `X`, not the number of possible variable bindings of `X` is evaluated, but instead `X`’s numerical value. If the variable list contains multiple elements, their values are multiplied to give the feature evaluation.

These transformations, however, cannot be easily applied to GDL games, since GDL lacks any definition of arithmetics. This is already partly solved in Fluxplayer, since the domains of a predicate’s arguments are calculated, as well as certain properties of predicates like transitivity, reflexivity and symmetry. If there is a total order between a set of domain elements, one could treat them as numbers. Then, one could deduce if a predicate that only contains numerical arguments if it represents one of a set of given mathematical calculations and comparisons, such as  $+$ ,  $-$ ,  $>$ ,  $\geq$ ,  $<$  or  $\leq$ .

In the present system, these two transformations have not been implemented, mainly because of a lack of time. Also, there are two other reasons:

1. In the case of multiple numerical variables in a feature’s variable list, the values are simply multiplied, as discussed above. This seems somewhat arbitrary (e.g., in the third generated feature in the example above, `C` should probably rather be divided by `D` instead of being multiplied).
2. There is no mention of how the feature evaluation for numerical features is defined in the case that the feature’s formula can be satisfied more than once. Presumably, Zenith uses the determinacy information that is required to be stated for all predicates in Zenith’s game description to only generate formulæ that cannot be resatisfied. If that is the case, this determinacy information would have to be automatically inferred from the GDL game description, since it does not contain this information.

Without these transformations, the system is not able to “understand” numbers and decompose features based on numerical values. The calculations and comparisons as such will be treated like any other predicate. However, since the necessary predicates are explicitly given in GDL, it can partly emulate this effect by applying other transformations. For example, the goal description of the game Asteroids contains the atom `true(north-speed(0))`. From this, the system can derive features with formula `true(north-speed(N))`, with  $N \in \{-3, \dots, +3\}$ .

Still, including these transformations in future versions of this system would be worthwhile, since they allow to represent higher-level concepts in just one feature, whereas the current implementation would need many features for the same task.

### Variable-Specialize

Zenith also contained another transformation called `variable-specialize`. It tries to find invariant variable values that always satisfy part of a feature’s formula, and generate a new feature that specializes on these values. More specifically, if a formula contains a single variable  $X$ , it is split into a prefix  $p$  that binds  $X$ , and a suffix  $q$ , which must be a unary atom that uses  $X$ . Then, the set  $S = \{X \mid p(X) \implies q(X)\}$  is computed. If this set is nonempty, `variable-specialize` creates a new feature with the formula  $p(X) \wedge (X \in S)$ . This set inclusion test is usually much cheaper than the replaced subformula  $q$ .

`variable-specialize` has deliberately not been implemented for this thesis. The reason for this is that due to the lack of an appropriate theorem prover, Fawcett had to use a test on a large number of states to approximate the invariance of variable values. In order for this test to be sufficiently reliable, one would have to use quite a large set of states, especially if the prefix  $p(X)$  is only true in a small fraction of all states. The resulting time needed to compute this transformation seemed too high to use it on the large number of features generated in the current implementation.

## 3.4 Simplifier

Whenever one of the feature transformations generates a new feature, it is passed to the simplifier, which applies various syntactic simplifications to the feature’s formula. The purpose of this is

1. to reduce the computation cost of the feature while maintaining logical equivalence,
2. to remove redundancies in the feature’s formula, so the efficiency of the duplicate removal phase (Section 3.6) is increased, and
3. to eliminate features with an unsatisfiable or state-independent formula.

The simplifier works through a series of steps. If the formula is determined to be unsatisfiable at one point, it is rejected; otherwise, the simplified formula is returned.

### Step 1: Expanding single-clause predicates

This step counts, for each atom  $A$  in the feature’s formula (except `true`), the number of clauses that the atom would match. If there is no such clause at all, it is replaced by `false`. If there is exactly one such clause  $H \Leftarrow B$ ,  $A$  is replaced by  $(A = H) \wedge B$ .

### Step 2: Removing unifications with singleton variables

In this step, all singleton variables (i. e., variables that occur nowhere else in the formula) are identified. All unifications with a singleton variable  $S$ , i. e. expressions of the form  $S = T$  or  $T = S$ , are replaced by *true*. Likewise, all expressions of the form  $S \neq T$  or  $T \neq S$  are replaced by *false*. Since this can produce new singleton variables, this step is repeated until no more replacements take place.

Note: To distinguish between the boolean truth value *true* and the special GDL predicate `true`, different fonts are used.

### Step 3: Evaluating unifications in conjunctions

All unifications  $T_1 = T_2$  in the outermost conjunction of the formula are evaluated (possibly binding some variables). If both terms are not unifiable, the formula is unsatisfiable. Otherwise, the unification is replaced by *true*.

### Step 4: Removing true and false

This step iteratively

- replaces  $\neg false$  by *true*,  $\neg true$  by *false*;
- removes *true* if it appears as a conjunct;
- removes *false* if it appears as a disjunct;
- replaces conjunctions that contain *false* by *false*; and
- replaces disjunctions that contain *true* by *true*.

### Step 5: Removing duplicate atoms

If a conjunction or disjunction contains the same term twice or more, all occurrences except the first are removed from the formula.

### Step 6: Checking for state-independency

If the resulting formula does not contain at least one state-dependent atom, the feature is rejected.

## 3.5 Trim-Variables

One of the key problems of feature generation is how to limit the number of generated features. The `trim-variables` heuristic described in this section is an extension to the feature generation algorithm designed to reduce this number by eliminating features that have the same output.

Since each of the initial features starts with a full variable list, feature generation – as described so far – will generate, for each formula, one version of the feature with every possible variable list. If the variable list has  $n$  variables, this means that

$2^n$  versions of this feature will be generated. Many of these features have the same evaluation on all states. The idea behind `trim-variables` is to reduce this set to only those features which have a different evaluation.

The actual `trim-variables` heuristic is quite simple: Whenever a new feature is generated, `trim-variables` tries to remove one of its variables from the variable list. If the resulting feature still has the same evaluation, the procedure is repeated until none of the remaining variables can be removed.

Unfortunately, proving that a variable can be safely removed without changing the evaluation in any reachable state would in general require to traverse the whole state space. Therefore, a heuristical approach has been chosen, in which the evaluations of the two features, the old and the new one, are compared on a random set of states. This set is generated before starting the feature generation algorithm by randomly traversing the state space until  $n_1$  unique non-terminal states have been collected, and then randomly picking  $n_2$  states from this set. In the experiments conducted for this thesis, the parameters  $n_1 = 3000$  and  $n_2 = 100$  were used. The reason for this particular implementation is that the larger set will be used by the feature selection algorithm later on to compute some metrics on the features.

Of course, using observations – especially on such a small set – instead of formal methods is error-prone, and only a last resort in absence of a better alternative. However, there are some arguments why the impact of this inaccuracy is limited, and the heuristic performed quite well in practical experiments:

1. The two compared features only differ in their variable list. This implies that they match (i. e., have an evaluation  $\neq 0$ ) in exactly the same states.
2. If a feature is too special or too general, i. e., it matches in very many or very few states, it will be filtered out by feature selection.

These two points imply that for any feature that has a chance of being selected later, some observations will probably be made, and they will be made on the same states. If the two features have the same evaluations on all of these observations, the differences will be minor, and it will probably be a good strategy to select only one of them for the evaluation function, even if they should not be logically equivalent.

Another aspect is that there is a compromise between speed and accuracy of this heuristic. The accuracy could be improved, at the expense of speed, by increasing the number of tested states.

It is easy to verify that when using the `trim-variables` heuristic, feature generation actually generates at most as many features as when the extension is not used: A feature is eliminated before it is added to the feature set and replaced by a version of the same feature with a reduced variable list. This feature will also be generated by the algorithm without the extension by application of the `remove-variable` transformation.

To ensure that any feature that is generated without the extension can also be generated with it (except for those that have equal evaluations, of course), some of the transformations have to be modified. These transformations need to restore a full variable list to each generated feature. An example that demonstrates the need for this is the following: Assume a feature has a formula that is unsatisfiable. `Trim-variables` will reduce the variable list of this feature to the empty list, since removing any variable does not change the feature's evaluations. By application of abstraction transformations like `remove-conjunct`, a feature can be generated that has a satisfiable formula. Versions of this feature with a non-empty variable list can only be produced if `remove-conjunct` restores the variable list of all features it produces. The same reasoning applies to the two abstraction transformations `split-indep-conjunctions` and `remove-conjunct`, which also generate features that are more general than the original feature, and also possibly remove restrictions on the variables. `Remove-negation` and `regress-formula` are neither specialization nor abstraction transformations, so the features generated by them can also place less restrictions on the variables. This is why the variable list has to be restored by these two transformations, too. On the other hand, the specialization transformations `split-disjunction`, `expand-to-base-case` and `expand-predicate` restrict the solutions to a feature's formula, and so does `remove-variable`. This makes restoring the variable list for these transformations unnecessary.

## 3.6 Duplicate Removal

In the final phase of the feature generation algorithm, each generated feature is checked if it has been generated before. To do so, the `DUPLICATEREMOVAL` algorithm calculates a key from the feature and uses this key for lookup in a hashtable containing all previously generated features. If a match is found, the new feature is discarded.

The key is computed in the following way:

1. Each variable in the feature's formula  $F$  is instantiated with the constant  $\#i$ , if the variable is the  $i$ th variable occurring in the formula, from left to right. This gives the instantiated formula  $F_{inst}$ .
2. The elements of the formula are sorted according to the Prolog standard ordering of terms, recursing into conjunctions, disjunctions and negations.
3. The instantiated variable list  $\vec{v}$  of the feature is sorted in the same way, giving  $\vec{v}_{inst}$ .
4. The key is the pair  $\langle F_{inst}, \vec{v}_{inst} \rangle$ .



This key is not unique, if two features contain the same subformulæ in a different order. For example, the algorithm would compute the following mappings for two semantically equivalent features:

$$\begin{aligned} \langle (p(X), q(Y)), [X, Y] \rangle &\longmapsto \langle (p(\#1), q(\#2)), [\#1, \#2] \rangle \\ \langle (q(Y), p(X)), [Y, X] \rangle &\longmapsto \langle (p(\#2), q(\#1)), [\#1, \#2] \rangle \end{aligned}$$

The sorting step of the algorithm only helps in those cases where the variables of the permuted subformulæ have already been bound, i. e., if these variables have previously occurred in the same order in both feature's formulæ. In general, detecting these homomorphisms would require to test all  $n!$  permutations of the  $n$  variables in the formula, which is not worth the effort. In practice, features with such permuted formulæ are very rare.

### 3.7 Restricting the Transformations

Originally, it was intended to use as few restrictions as possible during the feature generation phase, and use the feature selection algorithm to narrow down the set of features that will be passed to the learning algorithm. However, initial experiments showed that the feature transformations as described in the previous sections still produce an unmanageable number of features. Thus, the feature generation process has to be restricted more aggressively.

After extensive experimentation on a wide selection of game descriptions, the following restrictions were chosen:

1. **All transformations:** The maximum number of state-dependent atoms allowed to occur in any formula is eight. If at any point a feature is generated whose formula has more than eight state-dependent atoms, it is discarded.
2. **remove-conjunct:** This transformation is not applied to features that always match, i. e., have an evaluation of 1 for all states. Since **remove-conjunct** is an abstraction transformation, all features generated from such always-matching features will also match all states and therefore will provide no information about the quality of a state. The information whether a feature matches all states or not is computed by sampling the same set of states used in **trim-variables** (Section 3.5).
3. **remove-variable:** Like **remove-conjunct**, **remove-variable** will not be applied to features that match all states. Additionally, it will only be applied to features whose variable list contains five variables or less; if a feature contains more than five variables, all variables will be removed from the variable list.

Also, **remove-variable** checks all features that are generated from the same feature whether some of them have the same evaluations on all states, using the

same procedure as `trim-variables`, and only returns features with distinct evaluations.

4. `regress-formula`: This transformation only operates on features with less than four state-dependent atoms. Additionally, `regress-formula` cannot be applied more than three times during the whole derivation of the feature. Like `remove-conjunct` and `remove-variable`, it does not apply to features that match all states.
5. `expand-predicate`: `Expand-predicate` only expands atoms that have a maximum of four matching clauses.
6. `split-indep-conjunctions`: If `split-indep-conjunctions` is applicable, none of the other transformations will be executed on this feature.

The last restriction deserves a deeper explanation: Excluding all features whose formula consists of independent conjunctions reduces the number of generated features dramatically. This can be shown by a simple complexity analysis. Suppose a feature's formula consists of  $n$  independent subformulæ, and suppose that applying all transformations to each of these subformulæ would produce  $m$  new features on average. Then, the total number of features created from the original feature would be  $O(n^m)$ , whereas the number of features created after applying `split-indep-conjunctions` would only be  $O(n \cdot m)$ .

Apart from reducing the number of generated features, excluding features with independent subformulæ has another positive effect: Most of these features capture little additional information compared to the sum of the split subfeatures, but are much more expensive. The reason is that in many cases, the independent subformulæ have conceptually nothing to do with each other. If the variable list is non-empty, the number of possible instantiations of all features multiply, which makes the resulting feature very expensive. An example from the Chess variant Endgame for such a feature is

```
<<(true(cell(C1, R1, b)) ^ true(cell(C2, R2, wr)) ^ clearrow(C3, C2, R2)
^ true(cell(C3, R2, bk)) ^ kingmove(C3, R2, C4, R4)), [C1, R1, C4, R4]>> .
```

This feature's formula consists of two independent parts: The first conjunct, `true(cell(C1, R1, b))`, simply counts the number of blank spaces on the board, obviously not a very good feature. The rest of the formula counts the number of cells (C4, R4) that the black king could move to when it is checked via a row by the white rook. Since the variable list contains all of C1, R1, C4 and R4, the maximum value of this feature is  $61 \cdot 8 = 488$ . Splitting both parts into two separate features improves both the accuracy and the cost of the second feature, while the first will be discarded in a later step.

So, prohibiting the application of other transformations to features where the transformation `split-indep-conjunctions` is applicable has two positive effects: reducing the number of generated features as well as eliminating many useless features. However, these positive effects come at a price. One drawback is that in some cases, the independent parts of a formula have interactions that are lost when split apart. Since the linear function model that is used for learning cannot model non-linear effects like conjunctions, information is lost, and the quality of the sub-features combined can be lower than the original feature's quality. Examples from Tic-Tac-Toe are the features

$$\langle (\text{true}(\text{cell}(\text{R}, 1, \text{x})) \wedge \text{true}(\text{cell}(\text{R}, 2, \text{x})) \wedge \text{true}(\text{cell}(\text{R}, 3, \text{b})) \wedge \text{true}(\text{control}(\text{xplayer}))), [\text{R}] \rangle$$

and

$$\langle (\text{true}(\text{cell}(\text{R}, 1, \text{x})) \wedge \text{true}(\text{cell}(\text{R}, 2, \text{x})) \wedge \text{true}(\text{cell}(\text{R}, 3, \text{b})) \wedge \text{true}(\text{control}(\text{oplayer}))), [\text{R}] \rangle .$$

Since the formulæ `true(control(xplayer))` and `true(control(oplayer))` contain no variables at all, they are split apart from the rest of the formula. However, the first feature means an immediate win for `xplayer`, while the second doesn't; experiments conducted during implementation have shown that the game-play performance for Tic-Tac-Toe increases when all features that are similar to these two are added to the used feature set.

Another game where applying `split-indep-conjunctions` exclusively causes problems is Eight-Puzzle. One of the first features that are generated from the goal description is

$$\langle (\text{true}(\text{cell}(1, 1, 1)) \wedge \text{true}(\text{cell}(1, 2, 2)) \wedge \text{true}(\text{cell}(1, 3, 3)) \wedge \text{true}(\text{cell}(2, 1, 4)) \wedge \text{true}(\text{cell}(2, 2, 5)) \wedge \text{true}(\text{cell}(2, 3, 6)) \wedge \text{true}(\text{cell}(3, 1, 7)) \wedge \text{true}(\text{cell}(3, 2, 8)) \wedge \text{true}(\text{cell}(3, 3, \text{b}))), [] \rangle .$$

Again, since all conjuncts are ground, this feature is immediately split into all nine parts, which prevents the creation of many good features.

## 4 Feature Selection

In this chapter, the developed feature selection method will be described. We will begin by giving the motivation for the current method and formalizing the notion of an abstraction graph (Section 4.1). The next section will present a sound and efficient method to derive such an abstraction graph (Section 4.2). We will go on to describe how this graph can be used to assign a level of abstraction to each feature (Section 4.3) and conclude with a description of the resulting feature selection method (Section 4.4).

### 4.1 Abstraction-Graph-Based Feature Selection

The task of feature selection is to decide which of the generated features are included in the evaluation function. The main purpose of this is to limit the cost of the evaluation function. Even though some features can be excluded beforehand, based on several eligibility criteria (see Section 4.4 for details), there is still an overwhelming amount of features left. A method is needed to decide which of them should be selected. To answer this question, the following observation is helpful: The more special the features in an evaluation function get, the more “detailed” the overall evaluation function becomes. Very abstract features can only capture the rough overall objective of a game, while more special features can identify the fine points of play.

This motivates the idea of *abstraction-graph-based feature selection*: Start with the most abstract features, then add as many as possible of the more special ones until the time limit set for the evaluation function is reached. The more time one allows for the evaluation function, the more detailed it becomes. This approach helps to avoid concentrating on overly detailed features and completely missing a whole aspect of the game.

In order to arrive at a method that can quickly find the most abstract features, we will first formalize the terms “abstract” and “special”, and then introduce the notion of an abstraction graph.

**Definition 4.1** (Abstraction and Specialization). *A feature  $\varphi^a$  is called **more abstract** than another feature  $\varphi^s$ , iff in all reachable states  $z$ ,  $eval_{\varphi^s}(z) > 0 \implies eval_{\varphi^a}(z) > 0$ . Conversely,  $\varphi^s$  is called **more special** than  $\varphi^a$ .*

**Definition 4.2** (Detail). A feature  $\varphi_1$  is called **more detailed** than another feature  $\varphi_2$ , iff in all reachable states  $z$ ,  $eval_{\varphi_1}(z) > 0 \equiv eval_{\varphi_2}(z) > 0$  and  $eval_{\varphi_1}(z) \geq eval_{\varphi_2}(z)$ .

These definitions allow to introduce the notion of an abstraction graph, which we will later use to assign a level of abstraction to each feature.

**Definition 4.3** (Abstraction Graph). A graph  $\langle V, E \rangle$  is called an **abstraction graph**, iff

- all elements of  $V$  are features, and
- $\langle \varphi_1, \varphi_2 \rangle \in E \implies \varphi_1$  is either more abstract or less detailed than  $\varphi_2$ .

## 4.2 Building the Abstraction Graph

Unfortunately, building a *complete* abstraction graph for a given set of features would require to prove whether Definitions 4.1 and 4.2 hold for each pair of features in the set. This would, in turn, generally require to traverse the entire state space.

However, there is an alternative: We know that the features in the set were generated by the feature transformations from the previous chapter. By proving some properties of these transformations, and observing which of the features were generated by what transformation, it is possible to efficiently compute a *partial* abstraction graph.

**Lemma 4.1.** *Each feature that is generated by one of the abstraction transformations **split-indep-conjunctions** or **remove-conjunct** is more abstract than the feature it was generated from.*

*Proof.*

- **split-indep-conjunctions:** Let  $\varphi^a = \langle F^a, \vec{v}^a \rangle$  be a feature that was generated from feature  $\varphi^s = \langle F^s, \vec{v}^s \rangle$ . Then,  $F^s$  must have the form  $F_1 \wedge \dots \wedge F_n$ , and  $\exists i. (F_i = F^a)$ . Therefore  $F^s \implies F^a$ .
- **remove-conjunct:** Let  $\varphi^a = \langle F^a, \vec{v}^a \rangle$  be a feature that was generated by either from feature  $\varphi^s = \langle F^s, \vec{v}^s \rangle$ . Then,  $F^s$  must be equivalent to  $F^a \wedge F'$  for some formula  $F'$ , and  $F^s \implies F^a$ . □

**Lemma 4.2.** *Each feature that is generated by one of the specialization transformations **split-disjunction**, **expand-predicate** or **expand-to-base-case** is more special than the feature it was generated from.*

*Proof.*

- **split-disjunction:** Let  $\varphi^s = \langle F^s, \vec{v}^s \rangle$  be a feature that was generated from feature  $\varphi^a = \langle F^a, \vec{v}^a \rangle$ . Then,  $F^a$  must have the form  $F_1 \vee \dots \vee F_n$ , and  $\exists i. (F_i = F^s)$ . Therefore,  $F^s \implies F^a$ .
- **expand-predicate:** Let  $\varphi^s = \langle F^s, \vec{v}^s \rangle$  be a feature that was generated from feature  $\varphi^a = \langle F^a, \vec{v}^a \rangle$ . Then,  $F^a$  must have the form  $F_1 \wedge \dots \wedge F_n$ , and  $F^s$  must have the form  $G_1 \wedge \dots \wedge G_n$ . Furthermore, there is an  $i$  ( $1 \leq i \leq n$ ) with  $F_i \neq G_i$  and  $F_j = G_j$  for all  $j \neq i$ .  $F_i$  must be an atom that matches a clause  $H \Leftarrow B$ , and  $G_i$  must be  $(F_i = H \wedge B)$ . Therefore,  $F^s \implies F^a$ .
- **expand-to-base-case:** The proof is analogous to **expand-predicate**.  $\square$

**Lemma 4.3.** *Each feature that is generated by **remove-variable** is less detailed than the feature it was generated from.*

*Proof.* Let  $\varphi^a = \langle F^a, \vec{v}^a \rangle$  be a feature that was generated from feature  $\varphi^s = \langle F^s, \vec{v}^s \rangle$ . Since **remove-variable** does not change the formula,  $F^a = F^s$  and  $eval_{\varphi^a}(z) > 0 \equiv eval_{\varphi^s}(z) > 0$ . **Remove-variable** removes exactly one variable from  $\vec{v}^s$ , so  $eval_{\varphi^s}(z) \geq eval_{\varphi^a}(z)$  for all reachable states  $z$ , since each binding for variables in  $\vec{v}_s$  corresponds to at least one variable binding in  $\vec{v}^a$ .  $\square$

Lemmas 4.1–4.3 allow to build an abstraction graph by observing the feature generation process. To this end, the function **ADDTOABSTRACTIONGRAPH** (Algorithm 4.1 on the next page) is inserted immediately before the Duplicate Removal phase from Section 3.6. It receives two arguments as input:

- a triple  $\langle \varphi, t, \varphi' \rangle$ , where  $\varphi'$  is a new feature that has been generated from feature  $\varphi$  by transformation  $t$ , and
- an abstraction graph  $\langle V, E \rangle$ , initially empty.

The output is an updated abstraction graph that will be used as input of the algorithm for the next feature that it gets passed.

**Lemma 4.4.** *Each graph  $G = \langle V, E \rangle$  that was produced by Algorithm 4.1 has the following properties:*

1.  $G$  is an abstraction graph, and
2.  $G$  is a directed acyclic graph (DAG), i. e., if  $\langle \varphi, \varphi' \rangle \in E$ , then there is no path from  $\varphi'$  to  $\varphi$  in  $G$ .

*Proof.*

1.  $G$  is an abstraction graph:

This follows directly from the fact that for each edge  $\langle \varphi, \varphi' \rangle \in E$ , one of the following holds:

**Algorithm 4.1** AddToAbstractionGraph

---

```

1: function ADDTOABSTRACTIONGRAPH( $\langle \varphi, t, \varphi' \rangle, \langle V, E \rangle$ )
2:    $V' \leftarrow V \cup \{\varphi'\}$ 
3:   if  $t \in \{\text{split-indep-conj.}, \text{remove-conjunct}, \text{remove-variable}\}$  then
4:      $E' \leftarrow E \cup \{\langle \varphi', \varphi \rangle\}$ 
5:   else if  $t \in \{\text{split-disj.}, \text{expand-pred.}, \text{expand-to-base-case}\}$  then
6:      $E' \leftarrow E \cup \{\langle \varphi, \varphi' \rangle\}$ 
7:   else
8:      $E' = E$ 
9:   end if
10:  return  $\langle V', E' \rangle$ 
11: end function

```

---

- a)  $\varphi$  was generated from  $\varphi'$  by one of the abstraction transformations or the detailing transformation, in which case either Lemma 4.1 or Lemma 4.3 holds, or
- b)  $\varphi'$  was generated from  $\varphi$  by a specialization transformation, in which case Lemma 4.2 holds.

2.  $G$  is a DAG:

**remove-variable:** Since none of the other transformations changes the variable list, and each feature produced by **remove-variable** has a shorter variable list than the original one, there can be no cycles in  $G$  involving features generated by **remove-variable**.

**expand-predicate and expand-to-base-case:** For any given feature formula, **expand-predicate** and **expand-to-base-case** can only be applied a finite number of times ( $n$ ). The reason for this is that **expand-predicate** only expands non-recursive predicates, i. e., predicates which do not occur in a cycle in the dependency graph, and **expand-to-base-case** only expands to base cases, i. e., clauses which do not occur in a cycle in the dependency graph. This number  $n$  decreases with every application of **expand-predicate** or **expand-to-base**. Since none of the other abstraction, detailing or specialization transformations adds new atoms to the formula,  $n$  never increases. Therefore, there can be no cycle in  $G$  involving edges produced by either **expand-predicate** or **expand-to-base**.

**split-indep-conjunctions, remove-conjunct and split-disjunction:** All three remaining transformations produce formulæ that are shorter than the original one. Therefore, there can be no cycle among features produced by these three transformations.  $\square$

**Algorithm 4.2** TransitiveReduction

---

```
1: function TRANSRED( $\langle V, E \rangle$ : DAG)
2:    $E' \leftarrow E$ 
3:   for all  $v \in V$  do
4:     if  $\neg \exists v'. (\langle v', v \rangle \in E)$  then
5:        $e' \leftarrow \text{DFS}(\langle V, E' \rangle, v, \emptyset, \emptyset)$ 
6:     end if
7:   end for
8:   return  $\langle V, E' \rangle$ 
9: end function

10: function DFS( $\langle V, E \rangle$ : DAG,  $v$ : node,  $f$ : coming from node,  $S$ : search path)
11:   for all  $p$  with  $\langle p, v \rangle \in E$  do
12:     if  $p \in (S \setminus \{f\})$  then
13:        $E \leftarrow E \setminus \{\langle p, v \rangle\}$ 
14:     end if
15:   end for
16:   for all  $c$  with  $\langle v, c \rangle \in E$  do
17:      $E \leftarrow \text{DFS}(\langle V, E \rangle, c, v, S \cup \{c\})$ 
18:   end for
19:   return  $E$ 
20: end function
```

---

**Optimization: Transitive Reduction**

The performance of the following phases of feature selection can be slightly improved by calculating the transitive reduction of the abstraction graph.

**Definition 4.4** (Transitive Reduction). *The transitive reduction of a directed graph  $G = \langle V, E \rangle$  is the directed graph  $G' = \langle V, E' \rangle$  with the smallest number of edges such that for every path between vertices in  $G$ ,  $G'$  has a path between those vertices.*

Informally, transitive reduction can be thought of as the counterpart to transitive closure. For DAGs, the transitive reduction is unique. Algorithm 4.2 shows the implementation used for this thesis, based on depth-first search (DFS).

### 4.3 Assigning Abstraction Levels

The next task after computing the abstraction graph is to assign a *level* to each node to indicate how “high up” in the abstraction graph that node is. This enables selection of those features with the highest degrees of abstraction in the following section.



Algorithms for assigning levels to a DAG have previously been discussed in the context of drawing a visual representation of a graph. To formalize the notion of “level”, we will use a definition by Sugiyama, Tagawa, and Toda (1981):

**Definition 4.5** (*n*-level Hierarchy). *An n-level hierarchy ( $n \geq 2$ ) is defined as a directed graph  $\langle V, E \rangle$ , where  $V$  is called a set of vertices and  $E$  a set of edges, which satisfies the following conditions.*

1.  $V$  is partitioned into  $n$  subsets, that is

$$V = V_1 \cup V_2 \cup \dots \cup V_n \quad (V_i \cap V_j = \emptyset, i \neq j)$$

where  $V_i$  is called the  $i$ th level and  $n$  the length of the hierarchy.

2. Every edge  $e = \langle v_i, v_j \rangle \in E$ , where  $v_i \in V_i$  and  $v_j \in V_j$ , satisfies  $i < j$ , and each edge in  $E$  is unique.

The  $n$ -level hierarchy is denoted by  $G = \langle V, E, n \rangle$ .

So the task at hand is to convert the abstraction graph into an  $n$ -level hierarchy. However, there are many possible hierarchies corresponding to a given graph. For example, Figure 4.1 on page 43 shows two different hierarchies resulting from the same graph. Figure 4.1a shows a simple way of converting a graph to a hierarchy: The level assigned to each node is the longest distance from any source node (i. e., a node with no incoming edges; in the given graph, these are  $A$  and  $G$ ). However, the resulting hierarchy intuitively does not capture the desired notion of abstraction very well, since  $E$  and  $H$  are assigned different levels, although both are only one abstraction step away from  $F$ . Likewise,  $D$  and  $G$  are assigned vastly different levels, although both are only two steps away from  $F$ . Figure 4.1b, on the other hand, shows a hierarchy without the unnecessary long span edges  $\langle G, F \rangle$  and  $\langle G, H \rangle$ .

Such a hierarchy is computed by ASSIGNLEVELS (Algorithm 4.3). First, all nodes in  $V$  are sorted according to a topological ordering, which is done by the standard graph algorithm TOPSORT. A topological ordering is a list  $S$  of all nodes in a DAG  $\langle V, E \rangle$  with the property that, if there is an edge  $\langle v_1, v_2 \rangle \in E$ , then  $v_1$  occurs before  $v_2$  in  $S$ .

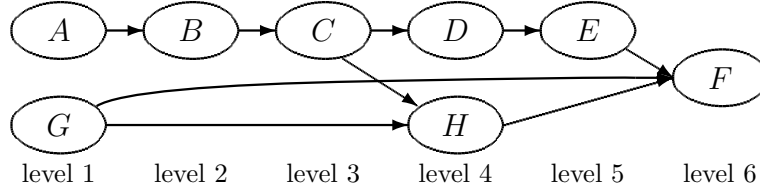
ASSIGNLEVELS uses this topological ordering to traverse the graph twice, once forward and once backward. In the first (forward) pass, the level assigned to each node is equal to its maximum distance from any source node. The purpose of this pass is to determine the correct level of all sink nodes (a sink node is a node without outgoing edges). In the second pass, the topological ordering is traversed backwards. In this step, the level assigned to each node is the minimum level of the node’s children. In the case of a sink node, the level computed in the first pass remains unchanged.

**Algorithm 4.3** AssignLevels

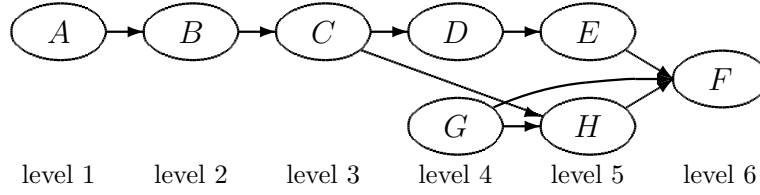
---

```
1: function ASSIGNLEVELS( $\langle V, E \rangle$ )
2:    $V_1 \leftarrow \emptyset, V_2 \leftarrow \emptyset, \dots, V_{|V|} \leftarrow \emptyset$ 
3:    $S \leftarrow \text{TOPSORT}(\langle V, E \rangle)$ 
4:   for all  $v \in S$  do
5:      $P \leftarrow \{p \mid \langle p, v \rangle \in E\}$ 
6:     if  $P = \emptyset$  then
7:        $V_1 \leftarrow V_1 \cup \{v\}$ 
8:     else
9:        $l_p \leftarrow \max(l)$ , where  $P \cap V_l \neq \emptyset$ 
10:       $V_{l_p+1} \leftarrow V_{l_p+1} \cup \{v\}$ 
11:    end if
12:  end for
13:
14:   $V'_1 \leftarrow \emptyset, V'_2 \leftarrow \emptyset, \dots, V'_{|V|} \leftarrow \emptyset$ 
15:   $R \leftarrow \text{REVERSE}(S)$ 
16:  for all  $v \in R$  do
17:     $C \leftarrow \{c \mid \langle v, c \rangle \in E\}$ 
18:    if  $C = \emptyset$  then
19:       $V'_l \leftarrow V'_l \cup \{v\}$  iff  $\{v\} \in V_l$ 
20:    else
21:       $l_c \leftarrow \min(l)$ , where  $C \cap V'_l \neq \emptyset$ 
22:       $V'_{l_c-1} \leftarrow V'_{l_c-1} \cup \{v\}$ 
23:    end if
24:  end for
25:   $n \leftarrow$  index of last non-empty set in  $V'_1, V'_2, \dots, V'_{|V|}$ 
26:  return  $\langle (V'_1 \cup V'_2 \cup \dots \cup V'_n), E, n \rangle$ 
27: end function
```

---



(a) A hierarchy where levels are assigned by the longest distance from any source node



(b) A hierarchy resulting from the AssignLevels algorithm

Figure 4.1: Two hierarchies of the same graph

**Lemma 4.5** (Correctness). *Let  $G = \langle V, E \rangle$  be a DAG, and  $\text{ASSIGNLEVELS}(G) = \langle (V'_1 \cup V'_2 \cup \dots \cup V'_n), E, n \rangle$ . Then,*

1.  $V = (V'_1 \cup V'_2 \cup \dots \cup V'_n)$ , and
2.  $\langle (V'_1 \cup V'_2 \cup \dots \cup V'_n), E, n \rangle$  is a  $n$ -level hierarchy.

*Proof.* We will show both conditions in turn.

1.  $V = (V'_1 \cup V'_2 \cup \dots \cup V'_n)$ : This follows from the fact that, in the first pass of the algorithm, each node is assigned to exactly one of the sets  $V_1, V_2, \dots, V_{|V|}$ , and in the second pass, each sink node carries over its level from these sets, and each non-sink node is assigned a new level, in exactly one of the sets  $V'_1, V'_2, \dots, V'_{|V|}$ . Also, since the level assigned to each sink node in the first pass is equal to the longest distance from any source node, all indices must be in the range  $1 \dots |V|$ .
2.  $\langle (V'_1 \cup V'_2 \cup \dots \cup V'_n), E, n \rangle$  is a  $n$ -level hierarchy:
  - a)  $V'_i \cap V'_j = \emptyset, i \neq j$ : This follows from the fact that exactly one level is assigned to each node, as stated above.
  - b) Every edge  $e = \langle v_i, v_j \rangle \in E$ , where  $v_i \in V'_i$  and  $v_j \in V'_j$ , satisfies  $i < j$ : Since the nodes are traversed in reverse topological ordering in the second pass, the level of  $v_j$  is calculated before  $v_i$ . The level assigned to  $v_i$  is smaller than the minimum of all its children, including  $v_j$ , so  $i < j$ .

- c) *Each edge in  $E$  is unique*: This follows from the fact that  $E$  is a set, and the edges are unlabeled.  $\square$

## 4.4 Final Feature Selection

During the final feature selection phase, features are selected for inclusion in the evaluation function. Starting with the most abstract level of the abstraction graph, features are added to the evaluation function until a given time limit for the evaluation function is reached. Before a feature is added, a number of precursory checks is performed to exclude features that will probably not improve the quality of the evaluation function.

Features are excluded based on the following principles:

1. Features with a *negated formula* are redundant: they provide no new information over the non-negated version.
2. Features that have *independent conjuncts* (i. e., can be processed by the transformation `split-indep-conjunction`) are usually much more expensive and less expressive than the separated single conjuncts (see Section 3.7 for a detailed discussion).
3. Features that are *too special* (i. e., match in no or almost no states) are costly to compute relative to their utility, since they have to be evaluated for each state, but only match in few of them. Also, they do not generalize well to other states, which introduces the risk of overfitting by allowing the evaluation function too many degrees of freedom.
4. Features that are *too general* (i. e., match in all or almost all states) are the converse case: They only provide information about the few states in which they do *not* match. Good features should be general enough to avoid overfitting, while being special enough to separate good from bad states.
5. Features that are *too expensive* take up a disproportionate amount of the overall computation time for the information they provide.

While criteria 1 and 2 can easily be computed based on syntactical analysis of the feature, criteria 3–5 require observation of the feature’s evaluations and the needed computation time on a set of states. The procedure is similar to the one described in Section 3.5 (Trim-Variables): The feature’s evaluations are observed on a set of non-terminal states that were collected using random game play, but this time using a much larger set of states (in the conducted experiments, a set of 3000 states was used). Afterwards, the *matching ratio* is computed as the number of states in which the feature’s evaluation was greater than 0 divided by the total number of observed

states. If the matching ratio is below a given threshold (currently 1%), it is rejected because it is too special. Likewise, if the matching ratio is above another threshold (currently 99%), it is rejected as being too general. The total time it took to compute all evaluations, divided by the total number of states, gives the *evaluation time per state*; if it exceeds a given percentage of the allowed total evaluation function time (currently 3%), it is rejected as being too expensive. Another value that is observed and stored here is the *maximum value* of the feature. This will later be used to normalize the feature's evaluation.

Since the abstraction graph allows the selection of the most abstract features before performing such tests, only a small percentage of all generated features has to be tested. This allows the usage of such a large set of states. Additionally, the time taken by this procedure can be sped up considerably by using a sequential statistical test procedure devised by Buro (1999). It is based on the observation that, if for example among the first 1000 tested states, the feature matches only a single one, it is very unlikely that the total matching ratio exceeds 1%. The proposed heuristic uses the fact that the expected number of matched states in a sequence of length  $d$  is  $dq$ , where  $q$  is the probability of a match, and the standard deviation is  $\sqrt{dq(1-q)}$ . The heuristic allows to set a confidence level  $t$ . When the statistical probability that the feature is too special or too general is higher than  $t$ , the function aborts early. In the conducted experiments,  $t$  was set to 95%. The resulting algorithm used to test whether a feature is eligible for inclusion in the evaluation function is shown in Algorithm 4.4 on the next page.

This algorithm can now be used to construct the final FEATURESELECTION algorithm (Algorithm 4.5 on page 47). It simply scans through the abstraction hierarchy, level by level, and checks each feature in the level if it is eligible for inclusion, until the total evaluation function time limit is reached. The final abstraction graph is passed to ASSIGNLEVELS again. This has the effect that, if parent nodes of a selected feature have not been selected, the feature will be assigned a lower level than before, thereby again removing unnecessary long span edges and “compacting” the abstraction hierarchy near the top.

The final resulting abstraction hierarchy is shown in Figure 4.2 on page 48<sup>1</sup>. One can clearly see that there are some features whose parents have been completely removed in the top portion of the figure, and that these features have been assigned to level 1. The graph has been colored by the matching ratio, which demonstrates the effect that the most abstract features are collected in the lower levels.

---

<sup>1</sup>The abstraction graphs that are computed for most games, including Tic-Tac-Toe, are too large to fit on one page and still be readable. Therefore, the graphs for all tested games are available in PDF format on the accompanying CD.

---

**Algorithm 4.4** FeatureEligible

---

```
1: function FEATUREELIGIBLE( $\varphi$ : feature)
2:   Constants:  $S$ : set of states,  $t$ : confidence level,  $l$ : time limit per state,
    $q_{min}, q_{max}$ : min/max matching ratio
3:   if  $\varphi = \langle (\neg F), \vec{v} \rangle$  then
4:     return false ▷ negated formula
5:   else if SPLIT-INDEP-CONJUNCTIONS( $\varphi$ )  $\neq \emptyset$  then
6:     return false ▷ formula has independent conjuncts
7:   end if
8:    $l_{max} \leftarrow l \cdot |S|$ 
9:    $u \leftarrow 0$ 
10:   $d \leftarrow 0$ 
11:  for all  $z \in S$  do
12:     $d \leftarrow d + 1$ 
13:    if  $eval_{\varphi}(z) > 0$  then
14:       $u \leftarrow u + 1$ 
15:    end if
16:    if  $u < dq_{min} - t\sqrt{dq_{min}(1 - q_{min})}$  then
17:      return false ▷  $Prob(\text{too special}) > t$ 
18:    else if  $u \geq dq_{max} + t\sqrt{dq_{max}(1 - q_{max})}$  then
19:      return false ▷  $Prob(\text{too general}) > t$ 
20:    else if CPU TIME()  $> l_{max}$  then
21:      return false ▷ evaluation time exceeded
22:    end if
23:  end for
24:  return true
25: end function
```

---

---

**Algorithm 4.5** FeatureSelection

---

```
1: function FEATURESELECTION( $\langle V_1 \cup V_2 \cup \dots \cup V_n, E, n \rangle$ : abstraction hierarchy,  
    $t_{max}$ : max evaluation function time)  
2:    $V' \leftarrow \emptyset$   
3:    $t \leftarrow 0$   
4:   for all  $V \in \{V_1, V_2, \dots, V_n\}$  do  
5:      $C \leftarrow \emptyset$   
6:     for all  $\varphi \in V$  do  
7:       if FEATUREELIGIBLE( $\varphi$ ) then  
8:          $C \leftarrow C \cup \{\varphi\}$   
9:          $t \leftarrow t + \text{EVALTIME}(\varphi)$   
10:      end if  
11:    end for  
12:    if  $t < t_{max}$  then  
13:       $V' \leftarrow V' \cup C$   
14:    else  
15:      break  
16:    end if  
17:  end for  
18:   $E' \leftarrow \{\langle v_1, v_2 \rangle \mid (\langle v_1, v_2 \rangle \in E) \wedge (v_1 \in V' \vee v_2 \in V')\}$   
19:  return ASSIGNLEVELS( $\langle V', E' \rangle$ )  
20: end function
```

---

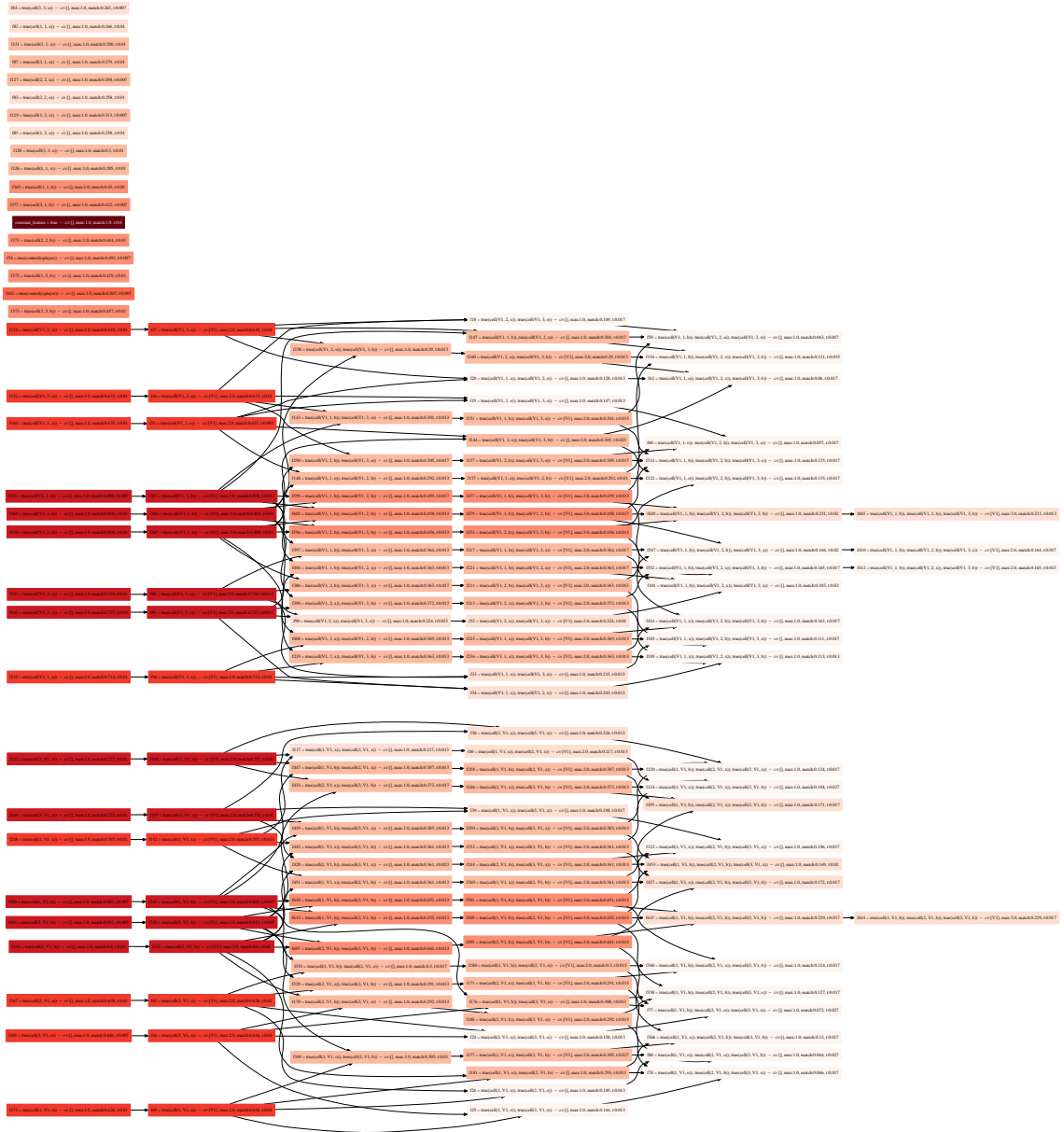


Figure 4.2: Abstraction graph for Tic-Tac-Toe. Features are colored by their matching ratio (darker shades mean higher ratio). All features that have the same level in the abstraction hierarchy are placed on the same horizontal rank.



## 5 Evaluation Function Learning

The purpose of the Evaluation Function Learning algorithm is to assign weights to the features that were selected in the previous chapter. The chosen learning method is TD( $\lambda$ ) reinforcement learning (Sutton and Barto, 1998).

The following four subsections are organized in a top-down order: First, we will consider the learning framework in which the training matches are run (Section 5.1). Next, the action selection strategies for both the learner and its opponents will be discussed (Section 5.2). Afterwards, the functional model for combining the features into an evaluation function will be explained (Section 5.3). The chapter concludes with a description of the specific TD weight update implementation (Section 5.4).

### 5.1 Learning Framework

The system learns the weights for each role of the game separately from playing a series of training matches (Algorithm 5.1 on the following page). In each state of the game, actions are selected for each role following an epsilon-greedy strategy by the function `SELECTACTION`<sup>1</sup> (Section 5.2). If the learner's action was selected greedily (i. e., the best action according to the current evaluation function has been selected), the weights are updated using the function `TDUPDATE` (Section 5.4). The variable  $\vec{e}$  stores the eligibility traces, which contain information on how often a feature has matched in the previous states and are used by the `TDUPDATE` algorithm. These are reset to  $\vec{0}$  whenever a random action is selected.

The weight vector  $\vec{w}$  is initialized randomly. The reason for this is that there exists no solution if all weights are initialized to the same value (for example 0) and the solution requires unequal weights, because the error is propagated backwards with a proportion of the weights. The random initialization serves to break symmetries.

### 5.2 Action Selection

The action selection for the learner's role during the test games (Algorithm 5.2 on page 51) follows an epsilon-greedy strategy. This means that with a small probability  $\epsilon$ , a random action will be selected using the function `SELECTRANDOMACTION` to ensure proper exploration of the state space.

---

<sup>1</sup>The function calls printed in `verbatim` are functions that refer to the game description.

**Algorithm 5.1** RunLearningMatches

---

```

1: function RUNLEARNINGMATCHES( $n$ : number of matches,  $r$ : learner role)
2:   for  $i = 1$  to  $n$  do
3:      $\vec{e} \leftarrow \vec{0}$  ▷ eligibility traces
4:      $\vec{w} \leftarrow \text{RANDOMVECTOR}([-0.005, +0.005])$ 
5:      $z \leftarrow \text{init}()$ 
6:     while  $\neg \text{terminal}(z)$  do
7:        $\langle \vec{m}, g \rangle \leftarrow \text{SELECTACTIONS}(r, z)$ 
8:        $z' \leftarrow \text{next}(\vec{m}, z)$ 
9:       if  $g = \text{true}$  then ▷ greedy action selected
10:         $\langle \vec{e}, \vec{w} \rangle \leftarrow \text{TDUPDATE}(r, \vec{e}, \vec{w}, z, z')$ 
11:       else ▷ random action selected
12:         $\vec{e} \leftarrow \vec{0}$ 
13:       end if
14:        $z \leftarrow z'$ 
15:     end while
16:   end for
17: end function

```

---

All  $(1 - \epsilon)$  other actions are selected greedily by the function `SELECTGREEDYACTION`. It performs a one-ply search, evaluating all successor states of the current state using the current evaluation function, and returning the action that leads to the highest-valued successor state. In the case of simultaneous games, `SELECTGREEDYACTION` picks a random action for all opponent roles before performing the one-ply search. Several alternatives to this have been considered:

1. Instead of assuming some random action for the opponents, one could directly use their evaluation function (as used by `SELECTOPPONENTACTION` below) to compute the action that they will select. However, this means that the learner is able to predict the opponents' actions perfectly, which is unrealistic in real game play and will probably cause the learner to overspecialize on the opponents used during training.
2. One could use the negated current evaluation function to predict the opponent's moves, and then compute a Nash equilibrium for all player's moves. The drawback would be that changing the weights in the evaluation function would not only change the learner's actions, but also its prediction of the opponents' actions. This would mean that the gradient of the evaluation function's weights is not clearly defined any more, which can have unpredictable effects on the learning task.
3. One could calculate the successor states for all joint moves and then average the expected rewards for each available action. While this method of action

**Algorithm 5.2** SelectActions

---

```

1: function SELECTACTIONS( $r$ : learner role,  $z$ : current state)
2:   Constants:  $\epsilon$ : exploration rate
3:    $\langle r_1, r_2, \dots, r_n \rangle \leftarrow \text{roles}()$ 
4:   for  $i = 1$  to  $n$  do
5:     if  $r_i = r$  then
6:       if  $\text{RANDOM}([0 \dots 1]) < \epsilon$  then
7:          $g \leftarrow \text{false}$ 
8:          $a_i \leftarrow \text{SELECTRANDOMACTION}(r_i, z)$ 
9:       else
10:         $g \leftarrow \text{true}$ 
11:         $a_i \leftarrow \text{SELECTGREEDYACTION}(r_i, z)$ 
12:       end if
13:     else
14:        $a_i \leftarrow \text{SELECTOPPONENTACTION}(r_i, z)$ 
15:     end if
16:   end for
17:   return  $\langle \langle a_1, a_2, \dots, a_n \rangle, g \rangle$ 
18: end function

```

---

selection would probably be more accurate than the one that was chosen, it would require to evaluate  $n_1 \cdot n_2 \cdot \dots \cdot n_r$  successors per selected action, where the  $n_i$  represent the number of available actions for each role, whereas the current method only has to evaluate  $n_l$  successors, where  $n_l$  is the number of actions available to the learner. Since this would slow the training games down considerably, the choice was made to use the current, less precise action selection method and make up for the imprecision by running a much greater number of training games. If the number of training games is large enough, the current method should approximate the averaging over actions on each step.

The actions of all roles that are not controlled by the learner are selected by the function `SELECTOPPONENTACTION`. Here, actions are selected greedily with random tie-breaking, using the Fluxplayer's current fuzzy-logic-based heuristic (Schiffel and Thielscher, 2007a,b). This ensures that the training games are played against a realistic opponent and speeds up the learning process compared to, for example, learning from self-play.

### 5.3 Evaluation Function Model

The functional model that is used for the evaluation function is the same that was used in Buro’s GLEM system: a linear combination of all features with a sigmoid squashing function.

Before the features’ evaluations enter the evaluation function, they are normalized. The normalized evaluation of feature  $\varphi_i$  in state  $z$  is defined as

$$evaln_{\varphi_i}(z) = \frac{eval_{\varphi_i}(z)}{max_{\varphi_i}}, \quad (5.1)$$

using the maximum value of the feature ( $max_{\varphi_i}$ ) that was computed in Section 4.4. This normalizing is not strictly necessary, since the same effect can be achieved by adjusting the feature weights, but it makes analysis and interpretation of the feature weights easier.

Using these normalized feature evaluations for the selected features  $\varphi_1, \varphi_2, \dots, \varphi_n$ , the evaluation function  $V(z)$  is defined as

$$V(z) = g \left( \sum_{i=1}^n w_i \cdot evaln_{\varphi_i}(z) \right), \quad (5.2)$$

where  $w_1, w_2, \dots, w_n \in \mathbb{R}$  are the feature weights, and  $g : \mathbb{R} \rightarrow \mathbb{R}$  is an increasing and differentiable link function.

This functional model, known in statistics as the generalized linear model, is identical to the standard linear model except for the link function  $g(x)$ . Its purpose is to restrict the output of the evaluation function to the possible output range. This could also be achieved by simply capping the output at these values; however, the resulting link function would not be differentiable, and saturation effects would arise.

The link function chosen here is widely used in neural networks and known as the sigmoid function:

$$g(x) = \frac{1}{1 + e^{-x}} - 0.5 \quad (5.3)$$

A nice property of this function is that its derivative, which will be needed by the TD weight update algorithm, is

$$g'(x) = \frac{1}{1 + e^{-x}} \cdot \left( 1 - \frac{1}{1 + e^{-x}} \right), \quad (5.4)$$

which can be quickly computed together with  $g(x)$ .

Plots of  $g(x)$  and  $g'(x)$  are shown in Figure 5.1 on the facing page. The weight change is greatest for the “undecided” states, where the output of the evaluation function is near 0. The closer the evaluation function’s output comes to saturation ( $-0.5$  or  $+0.5$ ), the smaller the weight change.

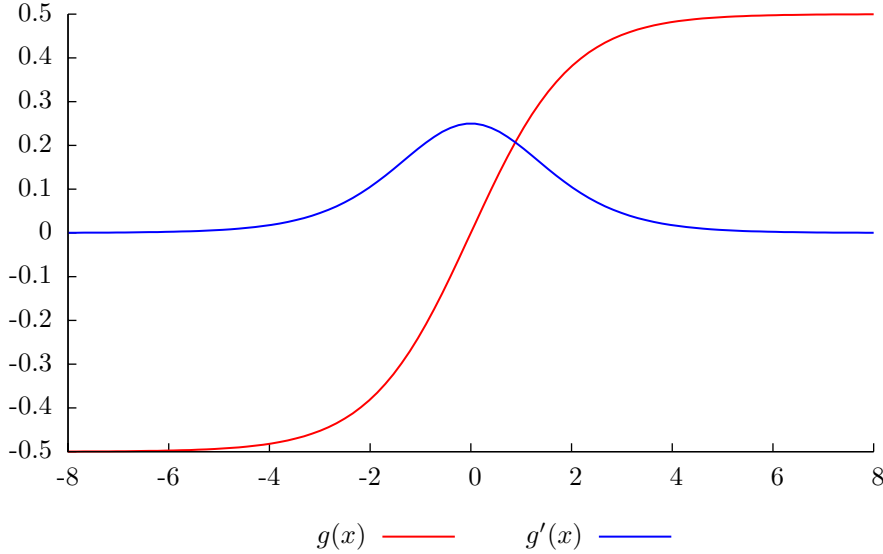


Figure 5.1: Plots of the link function  $g(x)$  and its derivative  $g'(x)$

---

### Algorithm 5.3 TD Update

---

- 1: **function** TDUPDATE( $r$ : learner role,  $\vec{e}$ : eligibility traces,  $\vec{w}$ : feature weights,  $z, z'$ : current & next state)
  - 2: Constants:  $\lambda$ : trace-decay rate,  $\gamma$ : future reward discount rate,  $\alpha$ : learning rate
  - 3:  $\delta \leftarrow R(r, z') + \gamma V(z') - V(z)$
  - 4:  $\vec{e}' \leftarrow \gamma \lambda \vec{e} + \nabla_{\vec{w}} V(z)$
  - 5:  $\vec{w}' \leftarrow \vec{w} + \alpha \delta \vec{e}'$
  - 6: **return**  $\langle \vec{e}', \vec{w}' \rangle$
  - 7: **end function**
- 

## 5.4 TD Weight Update

The weight update (Algorithm 5.3) is performed using the standard TD( $\lambda$ ) algorithm (Sutton and Barto, 1998).

The reward that the learning agent receives on each state is given by the function

$$R(r, z) = \begin{cases} 0.0096 \cdot (\text{goal}(r, z) - 50), & \text{if } \text{terminal}(z) \\ 0 & \text{otherwise} \end{cases}. \quad (5.5)$$

As one can see from this formula, the reward from the `goal` predicate is scaled from the interval  $[0, 100]$  into the interval  $[-0.48, 0.48]$ . The reason why 0.0096 has been chosen instead of 0.01 is that the link function  $g(x)$  has a domain of  $(-0.5, 0.5)$ , and if the outer limits of this domain are used for the training signal, it is possible that

there exists no optimal weight vector.

The gradient  $\nabla_{\vec{w}}V(z)$  consists of  $n$  partial derivatives, which can be computed separately for each  $i$  ( $1 \leq i \leq n$ ). Using  $g'(x)$  from Equation 5.3, this gives

$$\frac{\partial V}{\partial w_i}(z) = g' \left( \sum_{j=1}^n w_j \cdot \text{eval}n_{\varphi_j}(z) \right) \cdot \text{eval}n_{\varphi_i}(z), \quad (5.6)$$

so line 4 can be replaced by the following efficient iteration:

```
 $d \leftarrow g' \left( \sum_{j=1}^n w_j \cdot \text{eval}n_{\varphi_j}(z) \right)$   
for  $i = 1$  to  $n$  do  
     $e'_i \leftarrow \gamma \lambda e_i + d \cdot \text{eval}n_{\varphi_i}(z)$   
end for
```

## 6 Empirical Results

This chapter will deal with the results of empirical experiments that were conducted in this work to help judging the performance of the system. The order in which they will be presented follows the order of the preceding chapters: First feature generation and selection (Section 6.1), then evaluation function learning (Section 6.2).

### 6.1 Feature Generation and Selection

The system was tested on a diverse selection of 33 general games<sup>1</sup>, most of which have already appeared in one of the competitions. A short description of each game can be found in Appendix A.

All tests were performed on a 3.4 GHz Pentium IV with 2 GB of memory. The maximum time allowed for the evaluation function was 25 ms. This value is quite high, but has been chosen to allow the generation of many features and explore the system's abilities.

For some of the test games, the feature generation algorithm created too many or too few features. These games will be analyzed first, before the main results will be presented.

#### 6.1.1 Too Many Features

Feature generation was aborted after the 100,000<sup>th</sup> feature was expanded. This happened with the four most complex games: Blobwars and the three Chess variants Endgame, Minichess and Skirmish<sup>2</sup>. These games were excluded from all subsequent phases of the system (feature selection and evaluation function learning).

Figure 6.1 on page 57 demonstrates the development of the number of generated features for these games. For comparison, three games for which the Feature Generation algorithm successfully terminated are also included in the graph (Eight-Puzzle, Wallmaze and Merrills). An initial exponential growth in the number of generated features is clearly visible for all games; while the graphs for Minichess and Endgame already seem to flatten out, there is no sign of this for Blobwars and

---

<sup>1</sup>Most of these games are available at the GGP website: <http://games.stanford.edu/>.

<sup>2</sup>Note that, since the “parents” of each feature generation were counted instead of the “children”, the total number of generated features is actually higher than 100,000 for those games, as shown in Table 6.1 on page 60.

Skirmish. Their exponential growth clearly indicates that the restrictions placed on the feature transformations are not sufficient for these games.

### 6.1.2 Too Few Features

While the transformation restrictions were too loose for the games discussed in the previous subsection, it turns out that they were too strict for others.

After analysis of the generated features, five main reasons why the system could not generate sufficient features for some games could be identified:

#### Only recursive predicates in goal

This problem occurred in the games Hanoi and Othello. Hanoi’s goal predicate only contains the auxiliary predicate `tower` for counting the number of discs on the third tower. This predicate is recursive and is thus only expanded to its base case (zero discs), which prevents further expansion beyond the literal definition of the goal predicate.

The same problem happens with Othello. In Othello, there are two possible routes for feature development:

1. the calculation of goal values; this quickly leads to the predicate `piececount`, which is recursive, and
2. the calculation of legal moves, reached via regression; however, this leads to `openpiecestring` and `closedpiecestring` which are also recursive.

Unfortunately, all these recursive predicates only have very uninteresting base cases. Zenith could apply goal regression to these predicates, thanks to the goal regression information in Zenith’s domain theory; however, since GDL games do not contain this information, goal regression could not be applied here.

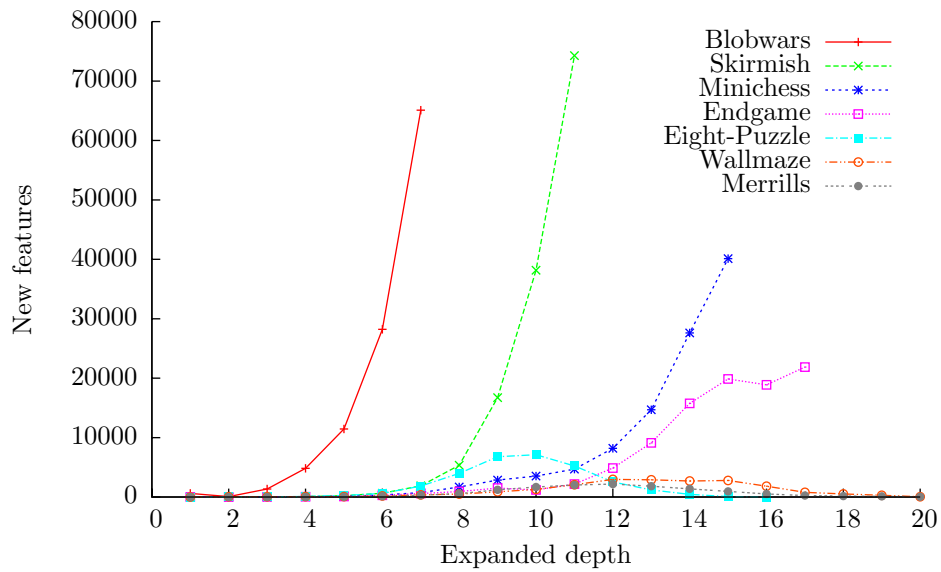
#### Goal formulæ too long

This is a problem for the games Quarto and Pentago. After expanding the macros and predicates in the goal description, all feature formulæ in both games are non-splittable conjunctions consisting of more than eight state-dependent conjuncts, so they are discarded even before any abstraction transformations are applied. This prevents the creation of any further features.

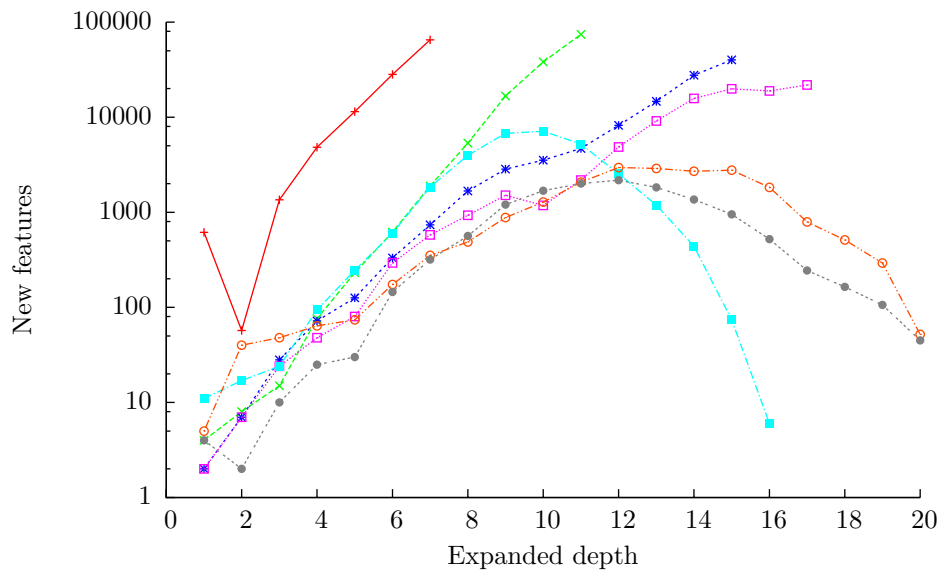
#### Non-descriptive goal predicate

The game Knightmove has a very non-descriptive goal predicate, since the reward only depends on the fluent `step`: The longer the player “survives”, the greater the reward. Since `step` is an ACF, it isn’t regressed, so no features exceeding the literal





(a) linear scale



(b) logarithmic scale

Figure 6.1: Number of new features per expansion depth for the four aborted games (Blobwars, Skirmish, Minichess, Endgame) and the three non-aborted games with the most features (Eight-Puzzle, Wallmaze, Merrills)

goal description are generated. Even if `step` was regressed, the generated features would probably not be very useful. Luckily, Knightmove is the only game where this problem occurred, since the goal descriptions of all other games contain some information on how to win the game.

### Missing arithmetics

The problem with Beatmania is that the player's points are accumulated in the fluent `blockscaught`, and only features based on that fluent would make sense. The feature formula `holds(blockscaught(V1), Z), scoremap(V1, V2)` from the goal description would have to be specialized (by expanding `scoremap`) to produce good features (otherwise it's always true), but since `scoremap` is not state-dependent (and also has 31 possible instantiations), it doesn't get expanded by `expand-predicate`. Alternatively, the variable `V1` could be treated as a number and used as the feature value (Fawcett's Zenith system contained a similar feature transformation called `split-arith-calc`). This would mean that one would have to detect that `scoremap` is a numerical function mapping the domain  $[0, 30]$  into the range  $[0, 100]$  and ergo `V1` and `V2` must also be numbers. While this is certainly feasible, it was not implemented due to lack of time.

### Good features too special

There were also two games, Pancakes and Mummy Maze, for which good features were generated by the feature expansion algorithm, but which were subsequently eliminated by feature selection because they were too special. In the game of Pancakes, all information is kept in a single fluent (`porder`) which represents a specific configuration of the pancakes. Through goal regression, all configurations that are 1–3 steps away from reaching this configuration are generated, which would certainly help in finding the solution. However, these features only match so seldomly in the testing states that they are discarded because they are too special.

For Mummy Maze, too, there is a set of good features, where the mummy is 1–3 steps away from catching the explorer, or the explorer is 1–3 steps away from reaching the exit, but these are also too special. The reason seems to be that the testing states are generated randomly, and the explorer doesn't find the exit through random movements often enough, and the mummy doesn't find the explorer often enough. Since (apart from the ACFs `control` and `step`) the game has only the fluent `location` (for the location of the mummy, explorer, and exit), all abstracted features become meaningless: If one of the `location` fluents is removed, the feature always matches and is filtered out by too-general.

Possible solutions to this problem for both games would be to either dynamically adjust the rate at which a feature is deemed too special, or to add another transformation that generalizes arguments of fluents, e. g., replacing constants by

variables. Another solution would be to use states from actual gameplay between expert players instead of random states for testing the matching ratio of features. Unfortunately, such expert matches are not readily available in GGP.

### 6.1.3 Successful Feature Generation

After removing those games for which either too many or too few features were generated, 21 games for which a good number of features were generated are left. On these games, the feature generation algorithm produced 14,509 features on average (range 48–118,016), of which 4,679 (42–30,184) are unique. Of those unique features, the feature selection step picked on average 143 (20–632) features for inclusion in the evaluation function. The resulting evaluation function took 10.18 milliseconds on average (0.69–21.16) to compute for each state. The detailed results for each game are shown in Table 6.1 on the following page. In most of the games, the total time needed for the evaluation function stayed well below the limit of 25 ms, because only full levels are included in the evaluation function; if the limit was exceeded during the calculation of one level, none of the features of that level were included.

The average time needed for the feature generation algorithm was 124.9 seconds (0.23–626.34); the feature selection algorithm took on average 216.31 seconds (2.97–1184.68). Table 6.2 on page 61 shows the detailed numbers for each game.

### 6.1.4 Cost of transformations

Figure 6.2 on page 64 shows the numbers of produced features and the needed computation time for each transformation. Since the computation time for a single transformation is usually smaller than the time resolution of the used timer, these results are somewhat unreliable: their sum is between 10% and 50% smaller than the total time when measuring the whole process. However, the rough proportions should be valid and can give an indication of the relative cost of the feature transformations.

The most expensive transformations are remove-conjunct and expand-predicate. Together, they account for more than 75% of the computation time. This can be explained by the fact that these two transformations have to restore the variable list (as described in Section 3.5), so `trim-variables` has to traverse a set of states, which is the most expensive part of feature generation.

Table 6.3 also shows that split-disjunction has never been executed. The reason for this is that while the game descriptions of 9 of the 33 test games contained disjunctions (Eight-Puzzle, Blobwars, Checkers, Endgame, Merrills, Minichess, Tic-Tac-Toe, Tic-Tic-Toe, Wallmaze), no valid feature was generated where this disjunction occurred at the top level of the feature formula. Either the disjunction contained no state-dependent subformula, or the feature contained non-state-dependent conjuncts besides the disjunction that were never removed.

Table 6.1: Number of generated, unique and selected features; average computation time needed for the invocation of the resulting evaluation function per state

	game	generated features	unique features	selected features	evaluation function time [ms]
too few features	Beatmania	155	79	23	0.25
	Hanoi	7	7	3	0.00
	Knightmove	14	14	6	0.11
	Mummy Maze	1010	553	7	2.05
	Othello	90	60	3	0.13
	Pancakes	386	325	10	0.09
	Pentago	20	12	1	0.00
	Quarto	17	12	3	0.02
good features	Asteroids	455	274	138	3.83
	Blocker	8510	1964	154	18.41
	Bombberman	3072	1171	122	17.36
	Breakthrough	14128	4730	133	8.92
	Checkers	5271	2756	632	21.16
	Chinese Checkers	48	42	25	1.94
	Circle Solitaire	243	148	123	19.45
	Connect-Four	803	273	141	13.12
	Crisscross	953	452	122	2.75
	Crossers3	12330	2871	122	19.68
	Eight-Puzzle	118016	30184	144	3.43
	Ghostmaze	9455	3963	78	19.71
	Hallway	1763	630	93	17.73
	Incredible	238	129	58	0.82
	Merrills	43467	13411	189	14.44
	Pacman	25634	10541	137	14.87
	Peg	1591	672	20	0.69
	Racetrack Corridor	1454	649	168	6.28
	Tic-Tac-Toe	640	254	151	1.95
Tic-Tic-Toe	10831	2838	219	4.57	
Wallmaze	45778	20308	29	2.67	
too many	Blobwars	391214	111598	—	—
	Endgame	276769	100317	—	—
	Minichess	284338	104659	—	—
	Skirmish	350321	137335	—	—

Table 6.2: CPU time needed for Feature Generation and Feature Selection

game	Feature Generation time [s]	Feature Selection time [s]	<i>total [s]</i>	
too few features	Beatmania	0.20	1.04	<i>1.24</i>
	Hanoi	0.01	0.01	<i>0.02</i>
	Knightmove	0.08	0.64	<i>0.72</i>
	Mummy Maze	13.37	97.94	<i>111.31</i>
	Othello	2.85	22.96	<i>25.81</i>
	Pancakes	0.30	1.16	<i>1.46</i>
	Pentago	0.32	0.12	<i>0.44</i>
	Quarto	0.05	0.12	<i>0.17</i>
good features	Asteroids	3.18	12.96	<i>16.14</i>
	Blocker	15.31	94.05	<i>109.36</i>
	Bombberman	229.08	656.36	<i>885.44</i>
	Breakthrough	122.92	95.29	<i>218.21</i>
	Checkers	40.89	269.12	<i>310.01</i>
	Chinese Checkers	0.23	6.17	<i>6.40</i>
	Circle Solitaire	1.40	58.85	<i>60.25</i>
	Connect-Four	3.17	42.40	<i>45.57</i>
	Crisscross	2.17	9.81	<i>11.98</i>
	Crossers3	23.63	98.36	<i>121.99</i>
	Eight-Puzzle	467.09	75.26	<i>542.35</i>
	Ghostmaze	216.31	600.84	<i>817.15</i>
	Hallway	34.86	127.00	<i>161.86</i>
	Incredible	0.25	2.97	<i>3.22</i>
	Merrills	626.34	654.82	<i>1281.16</i>
	Pacman	463.54	1184.68	<i>1648.22</i>
	Peg	5.82	47.78	<i>53.60</i>
	Racetrack Corridor	6.66	26.32	<i>32.98</i>
Tic-Tac-Toe	0.56	6.73	<i>7.29</i>	
Tic-Tic-Toe	26.06	69.08	<i>95.14</i>	
Wallmaze	333.41	403.72	<i>737.13</i>	
too many	Blobwars	1414.20	—	—
	Endgame	7736.25	—	—
	Minichess	2610.22	—	—
	Skirmish	5832.62	—	—

Table 6.3: Total number of features produced by each transformation

	game	remove- conjunct	split- indep- conj.	expand- predicate	regress- formula	expand- to-base- case
too few features	Beatmania	5	93	21	24	0
	Hanoi	0	0	0	0	0
	Knightmove	0	0	0	0	0
	Mummy Maze	308	565	108	21	0
	Othello	29	24	2	2	25
	Pancakes	0	22	0	350	0
	Pentago	0	8	2	0	0
	Quarto	0	8	1	0	0
good features	Asteroids	159	159	0	108	0
	Blocker	4538	695	0	1272	0
	Bombberman	1210	1266	0	572	0
	Breakthrough	8722	1870	1348	199	0
	Checkers	854	2744	8	1254	0
	Chinese Checkers	0	27	3	9	0
	Circle Solitaire	20	73	3	61	0
	Connect-Four	270	109	6	282	0
	Crisscross	0	608	215	110	0
	Crossers3	594	9359	6	2347	0
	Eight-Puzzle	72683	6503	24385	3662	0
	Ghostmaze	3501	4125	1308	400	0
	Hallway	1032	475	2	62	0
	Incredible	70	104	0	44	0
	Merrills	18120	18924	2456	2902	0
	Pacman	9669	11481	4046	420	0
	Peg	1450	10	0	102	0
	Racetrack Corridor	438	728	12	221	0
	Tic-Tac-Toe	234	187	10	154	0
	Tic-Tic-Toe	3943	20	10	890	0
Wallmaze	22260	14902	7104	860	0	
too many	Blobwars	3561	237258	2593	146841	0
	Endgame	125708	29590	34800	25906	49606
	Minichess	141527	18436	93522	21783	0
	Skirmish	127917	47185	112519	32572	20100

Table 6.3: (cont.) Total number of features produced by each transformation

	game	remove- variable	make- root- features	split- disjunction	remove- negation	<i>total</i>
too few features	Beatmania	4	4	0	4	155
	Hanoi	0	7	0	0	7
	Knightmove	0	13	0	1	14
	Mummy Maze	0	6	0	2	1010
	Othello	0	6	0	2	90
	Pancakes	0	14	0	0	386
	Pentago	0	7	0	3	20
	Quarto	0	7	0	1	17
good features	Asteroids	22	6	0	1	455
	Blocker	2000	4	0	1	8510
	Bombberman	10	9	0	5	3072
	Breakthrough	1982	5	0	2	14128
	Checkers	404	7	0	0	5271
	Chinese Checkers	3	6	0	0	48
	Circle Solitaire	80	6	0	0	243
	Connect-Four	127	6	0	3	803
	Crisscross	15	5	0	0	953
	Crossers3	3	14	0	7	12330
	Eight-Puzzle	10777	6	0	0	118016
	Ghostmaze	109	7	0	5	9455
	Hallway	186	6	0	0	1763
	Incredible	6	11	0	3	238
	Merrills	1056	8	0	1	43467
	Pacman	7	9	0	2	25634
	Peg	13	15	0	1	1591
	Racetrack Corridor	46	9	0	0	1454
	Tic-Tac-Toe	47	5	0	3	640
	Tic-Tic-Toe	5958	8	0	2	10831
Wallmaze	640	10	0	2	45778	
too many	Blobwars	953	7	0	1	391214
	Endgame	11151	5	0	3	276769
	Minichess	9062	5	0	3	284338
	Skirmish	10021	7	0	0	350321

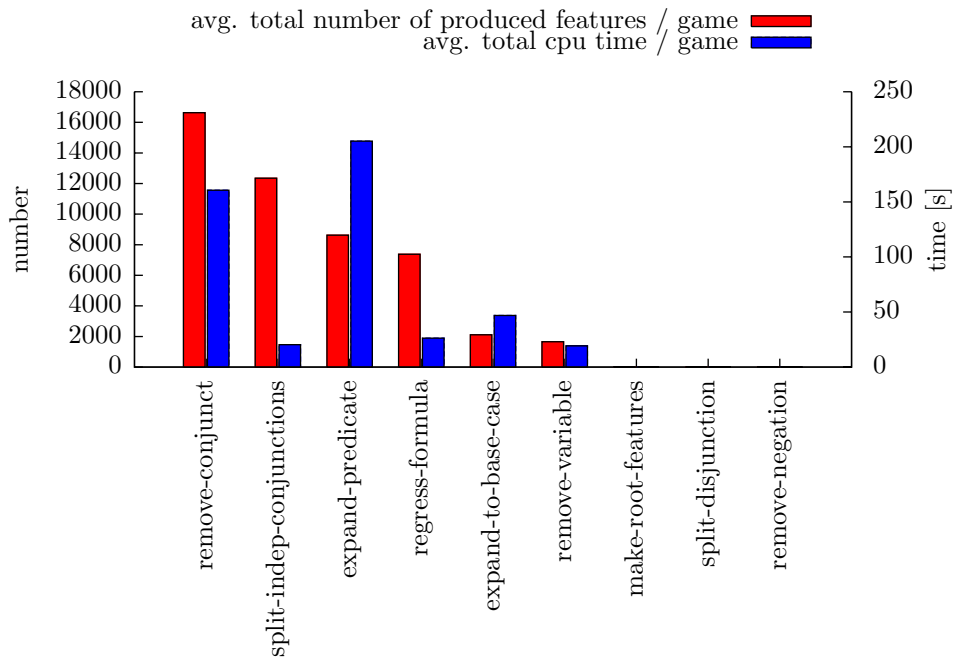


Figure 6.2: Number of features produced and CPU time spent by each feature transformation, averaged over all 33 games

## 6.2 Evaluation Function Learning

Sections 6.1.1 and 6.1.2 have identified some games for which too many resp. too few features have been generated. These were excluded from further examination. On the remaining 21 games, the evaluation function learning algorithm has been run, playing 2000 training matches. Table 6.4 on the next page shows the average computation time needed to play a complete training match for each game. The training graphs are listed in Appendix B.

The parameters used for the learning algorithm were:

- learning rate:  $\alpha = 0.15$
- trace-decay parameter:  $\lambda = 0.7$
- future reward discount rate:  $\gamma = 0.9$
- exploration rate:  $\epsilon = 0.1$

After the training was completed, 400 evaluation matches were run, with the weights frozen and the epsilon-greedy parameter  $\epsilon$  set to 0, so that the learner always picked the action with the greatest expected reward. The opponents for



Table 6.4: Time needed for one training match

game	time [s]
Asteroids	0.75
Blocker	14.98
Bombberman	41.48
Breakthrough	22.34
Checkers	31.47
Chinese Checkers	4.27
Circle Solitaire	0.70
Connect-Four	6.64
Crisscross	1.66
Crossers3	5.94
Eight-Puzzle	0.74
Ghostmaze	20.45
Hallway	24.60
Incredible	0.83
Merrills	28.35
Pacman	7.94
Peg	1.43
Racetrack Corridor	2.02
Tic-Tac-Toe	0.59
Tic-Tic-Toe	2.52
Wallmaze	1.41

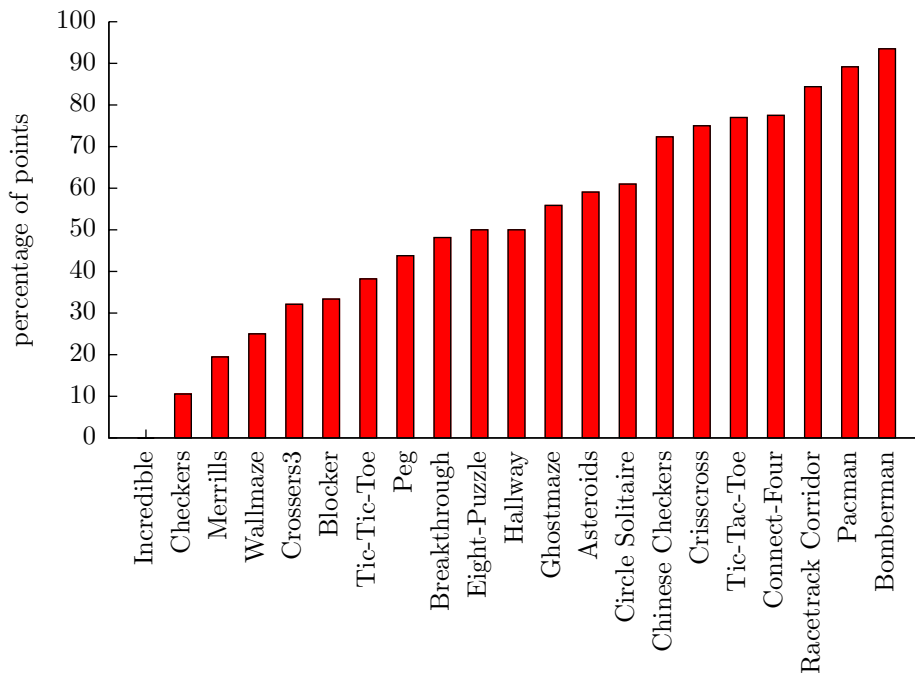


Figure 6.3: Percentage of average reward received by the learner vs. average of opponents

these matches were controlled by a single-ply search using Fluxplayer’s fuzzy-logic evaluation function.

The percentage of the reward that the learner received is displayed in Figure 6.3; the detailed results are listed in Table 6.5 on the next page. The opponent results listed for the single-player games were obtained by playing the game using the same opponent evaluation function used during multiplayer games. The following subsections will analyze the results in more detail. The games are grouped by their properties: single-player games (Section 6.2.1), turn-taking games, i. e., games in which all players except one have only one legal move in each state (Section 6.2.2), and simultaneous games (Section 6.2.3).

### 6.2.1 Single-Player Games

Two games could be identified for which no sufficient reinforcement signal was received: Eight-Puzzle and Incredible. All test matches of these games lead to a reward of 0. The reason is that before any weights have been learned, the player executes random actions, and the two games are so difficult that it is improbable to find a solution randomly.

Since the player never finds a solution, there is no useful feedback to the reinforcement learning algorithm. Thus, even though there may be good features in

Table 6.5: Average rewards after learning for 400 testmatches

game	learner role	learner	opponent 1	opponent 2
Asteroids	ship	50.00	34.63	—
Blocker	crosser	2.50	97.50	—
	Blocker	64.25	35.75	—
Bomberman	bomberman	89.13	10.88	—
	bomberwoman	97.88	2.13	—
Breakthrough	white	56.00	44.00	—
	black	40.25	59.75	—
Checkers	white	7.88	92.13	—
	black	13.25	86.75	—
Chinese Checkers	red	71.94	36.25	24.94
	green	80.38	24.94	25.00
	blue	74.44	24.94	37.13
Circle Solitaire	taker	100.00	63.96	—
Connect-Four	white	87.75	12.25	—
	red	67.25	32.75	—
Crisscross	red	25.00	0.00	—
	teal	0.00	0.00	—
Crossers3	top	15.75	42.15	34.60
	right	20.05	41.75	29.63
	left	17.28	33.03	42.98
Eight-Puzzle	player	0.00	0.00	—
Ghostmaze	explorer	49.88	50.13	—
	ghost	61.88	38.13	—
Hallway	white	50.00	50.00	—
	black	50.00	50.00	—
Incredible	robot	0.00	12.28	—
Merrills	white	25.81	74.19	—
	black	13.06	86.94	—
Pacman	pacman	43.31	15.50	0.00
	blinky	74.25	24.50	2.52
	inky	100.00	0.00	4.00
Peg	jumper	27.95	35.90	—
Racetrack Corridor	white	91.06	16.19	—
	black	91.34	17.54	—
Tic-Tac-Toe	xplayer	95.75	4.25	—
	oplayer	58.25	41.75	—
Tic-Tic-Toe	white	34.75	65.25	—
	black	41.63	58.38	—
Wallmaze	white	0.00	0.00	—
	black	0.00	100.00	—

the evaluation function, the system is unable to learn any weights. A possible solution would be to bootstrap the learning process from some “expert games” (see Section 7.1.2 for a discussion of this). This would perhaps allow the algorithm to assign the feature weights in a way that encourages moving towards the goal and thereby enable the system to learn further weights on its own.

The three remaining single-player games are Peg, Circle Solitaire and Asteroids. In Peg, around 44 % of the total reward (compared to Fluxplayer) could be achieved. As Figure B.32 on page 97 shows, the training graph shows no improvement, which is a hint that no good features could be generated from the game description.

For both Asteroids and Circle Solitaire, the learning system performs better than Fluxplayer (around 60 % of the total points). In the game of Asteroids, the system quickly learns to stop the ship and get 50 points; however, it doesn’t learn to stop the ship at the planet to receive the full reward of 100 points. Since Circle Solitaire is very simple, the system quickly learns a perfect winning strategy for this game.

## 6.2.2 Turn-Taking Games

This group of games comprises Breakthrough, Chinese Checkers, Crisscross, Tic-Tac-Toe, Connect-Four, Checkers and Merrills.

The performance on Checkers and Merrills was significantly lower than Fluxplayer’s (11 % of total points for Checkers, 19 % for Merrills). Analysis of the features generated for Checkers revealed that the complete feature set contained many similar features that only differed in various calls of the `minus1`, `minus2` and `minus3` binary predicates, which state that the first argument is a number that is 1 (resp. 2 or 3) greater than the second. In many of these calls, one of the variables was a singleton. Useless features of this kind could be eliminated by extending the simplifier.

The training graphs for Merrills (Figures B.27 and B.28 on page 95) show a strange phenomenon: The rewards actually diminish as learning progresses. This indicates a problem with the learning algorithm, which could not be clearly identified. Possibly the chosen learning rate was too high.

On the remaining five games, the system performed very well (48 % for Breakthrough, 72 %–78 % for the other four games). One notable effect occurred during the training of Crisscross: The Fluxplayer opponent always took the center of the board and stopped moving into its goal positions, blocking the learner. Figure 6.4 on the facing page shows the initial position and the final position, resulting in 25 points for red and 0 points for teal when Fluxplayer played teal. When Fluxplayer played red, it moved in a way that only one teal peg reached its goal, and moved a red peg in the last move of the game, resulting in 0 points for both players.

Such behavior would probably not occur during actual game play. This demonstrates one risk of using a fixed opponent for training: It is possible that the training matches bear little resemblance with actual game play, which can mislead the

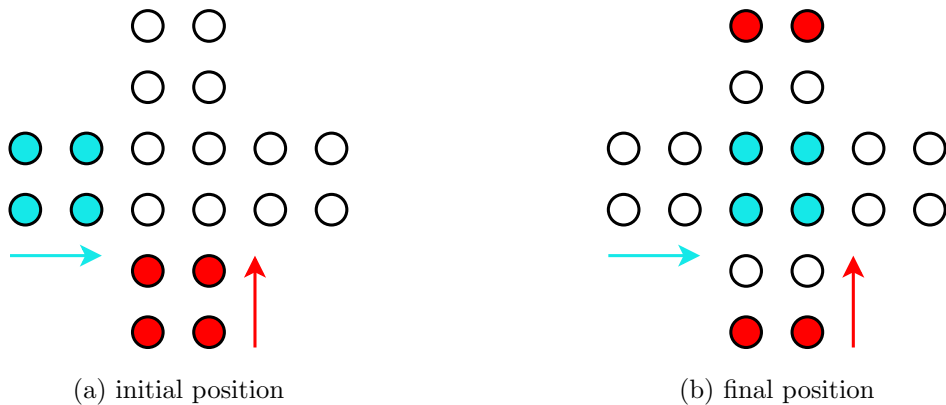


Figure 6.4: Initial and final positions of the game Crisscross

trained evaluation function. This problem could be alleviated by using self-play (playing against a backed-up version of the learning system itself).

### 6.2.3 Simultaneous Games

The last group of games consists of the games Hallway, Wallmaze, Crossers3, Blocker, Tic-Tic-Toe, Racetrack Corridor, Ghostmaze, Pacman and Bomberman.

In the games of Wallmaze and Hallway, all training matches ended with the same reward (0 for Wallmaze, 50 for Hallway). This meant that no meaningful reinforcement signal was received. The cause is similar to the single-player games Incredible and Eight-Puzzle: it is improbable to find a solution by executing random actions.

In Wallmaze, the goal is to exchange places with the opponent. The maze has very narrow passages (see Figure 6.5 on the next page), and players can block each other, so it is very improbable that a series of random actions leads to the goal position during the limit of 30 steps. In fact, this never happened during the test games. In Hallway, the situation is similar: both players start out on opposite directions of the board, placing walls that obstruct movement. In Hallway, too, there is the strict limit of 30 steps, and finding the solution randomly during that time limit is improbable. As discussed in Section 6.2.1, a solution could be to initialize the weights using expert games.

The games on which the system performed only moderately well are Crossers3 (32%), Blocker (33%) and Tic-Tic-Toe (38%). For Crossers3, the training graphs do not show any improvement, which is an indication that the generated features are not very good. In fact, analysis of the features shows that the highest-valued features only capture the notion of being one step away from the goal position, while Fluxplayer can use a full distance heuristic.

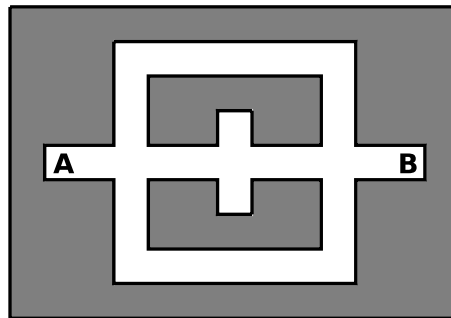


Figure 6.5: Initial state of the game Wallmaze

For Blocker and especially Tic-Tic-Toe, the low result is surprising, however: Many meaningful features could be generated. Since the game descriptions and generated features of Tic-Tic-Toe and Tic-Tac-Toe are very similar, and the results for Tic-Tac-Toe were much better, the reason is probably that the learner assumes a fixed random move for the opponent, upon which the learner can pick its move. Especially for the two games of Blocker and Tic-Tic-Toe, this assumption is unrealistic, because the outcome of an action is very strongly related to the other player's action. That is why the learner is seriously disadvantaged opposed to Fluxplayer on these games. The proper (but much more expensive) solution to this problem would be to compute an optimal strategy based on Nash equilibria, or at least use the same search strategies for both the learner and the opponent for the evaluation games.

This problem did not occur on the other four simultaneous games: Ghostmaze (55%), Racetrack Corridor (84%), Pacman (89%) and Bomberman (94%). In these games, the actions of the players do not interact as strongly as in Tic-Tic-Toe and Blocker. This could explain why the learner could achieve the majority of points on those games.

## 7 Conclusions

In this chapter, we will first assess the problems faced during the implementation of the system and point out some directions for future work (Section 7.1). Afterwards, a summary of the contributions will be given (Section 7.2).

### 7.1 Critical Assessment and Future Work

#### 7.1.1 Feature Generation

##### Remove-Variable and Trim-Variables

In retrospect, the implementation of the feature generation algorithm would have been much easier if the variable lists had been managed in a different way. Instead of generating features with a full variable list and removing variables afterwards, all features could be generated with an empty variable list, and features could be added by a new transformation `add-variable`. In fact, the whole abstraction graph could be built from features with an empty variable list, and variables could only be added for those features that are selected for inclusion in the evaluation function.

This would also render the `trim-variables` optimization unnecessary; since more than 90% of the computation time for feature generation is spent during the state traversals required by `trim-variables`, this would speed up the feature generation algorithm considerably.

This course of action was even proposed by Fawcett (1993, p. 61). He decided against it because Zenith does not develop features with a low discriminability further; if a feature with an empty variable list proved invaluable, it would be discarded, although a version of the feature with a non-empty variable list could be valuable.

##### Remove-Conjunct

`Remove-conjunct` is the transformation that is most expensive, both in the number of generated features and total computation time, so it seems worthwhile to look for ways to limit the number of features it produces. Additionally, `remove-conjunct` often removes a conjunct that is essential to the feature, so the generated feature has a much smaller value than the original one. If expert matches were available (see Section 7.1.2), one could detect if the generated feature's correlation with the eventual game outcome is much lower than the one of the original feature. This

would make it possible to discard such meaningless features without developing them further.

### **Expand-Predicate and Expand-To-Base-Case**

For some games, no good features could be generated because the goal and terminal predicates of their game description only contained recursive predicates. In that case, the only applicable transformation is `expand-to-base-case`, which may not generate good features. One could allow `expand-predicate` to partly unroll such recursive predicates; however, this would increase the number of generated features considerably. If a different approach to feature selection was chosen which does not require exhaustive application of the transformations, this idea could be revisited.

### **Additional Transformations**

The generated evaluation function would probably benefit from the inclusion of more higher-level concepts, for example detection of arithmetics. This would make it possible to use Zenith's `split-arith-comp` and `split-arith-calc` transformations. One could also add more game-playing-specific transformations that understand concepts like distance on boards or material value of pieces. Fluxplayer's current fuzzy-logic-based evaluation function already contains such concepts, and that is probably the reason why it outperforms the presented system in the more complex games (e.g., Checkers or Merrills). The concepts described above can in principle also be generated by the presented system, but take many more features, whereas direct detection of such concepts would increase the efficiency of the evaluation function.

## **7.1.2 Abstraction-Graph-Based Feature Selection**

Abstraction-graph-based feature selection has been developed as a way to select features without any information about the game except the game rules. Specifically, no expert matches are needed, which are not readily available in GGP. No a priori assessment of the quality of a feature is available, so the single selection criterion is the degree of abstraction of a feature.

While this resulted in a very fast feature generation and selection procedure, the experiments have unveiled several drawbacks:

1. In order to arrive at the most abstract features, the feature transformations have to be applied exhaustively. This imposed the need for very severe restrictions of the transformations. Still, the number of generated features was too high for the most complex games.
2. While these restrictions were too loose for some games, they were too strict for others, causing an insufficient number of features to be generated.



3. Using the degree of abstraction alone to guide the feature selection process causes many irrelevant features to be included in the evaluation function. These provide little or no information about the quality of a state while increasing the total cost of the evaluation function.

A general game player has to work on a large variety of games, and there is probably no fixed set of parameters that avoids both problem 1 and 2 for all games. This could be addressed by a dynamic transformation restriction procedure, which starts with very aggressive restrictions and loosens them until a sufficient number of features has been generated.

The impact of Problem 3 would be much smaller if a high number of features could be passed to the evaluation function learning algorithm, since the learning algorithm can efficiently assign near-zero weights to irrelevant features. Buro (1999) has demonstrated that fitting a huge number of feature weights ( $> 100,000$ ) is unproblematic. However, using so many features was only possible because these features are simple propositional conjunctions of atomic features. Such a restricted feature formalism allows the features to be evaluated very quickly once the atomic features have been evaluated. The actual set of atomic features used in GLEM was limited to 192 very simple features (“white”, “black” or “blank” for each board cell), which need to be known beforehand, so the actual feature selection takes place in the choice of the atomic features.

In the feature formalism used in this thesis, the limiting factor is the relatively high cost of the generated features. The feature formulæ contain variables, so the formulæ cannot easily be decomposed into parts that can be computed separately. Thus, each new feature adds to the time needed for the total evaluation function. This forces the selection step to limit the number of features very aggressively, and less features than expected could be included in the final evaluation function.

Given the importance of feature selection, the overall game-play performance of the presented system could probably be largely improved by incorporating actual game-play experience in the selection process. There are two ways to do this without falling back to the iterative approach:

1. **Staged evaluation functions.** GLEM contained a remarkable way to generate the training set using the game tree search itself, without the requirement of additional expert matches. It partitioned the game into 13 stages by ply number, for each of which a separate evaluation function was learned. Exhaustive search was used to generate the labelled training data used to train the evaluation function for stage 13; this evaluation function was used to train stage 12, and so on.

This is helped by the fact that the stages of an Othello match can be so easily identified: Practically all Othello matches consist of 58–60 plies. Using the ply number to separate, for example, a Chess match would probably not work so

well; instead, the number of remaining pieces on the board could be used to this purpose. The adoption of this approach to GGP may be difficult, since one does not know *a priori* how to partition a particular game.

2. **Expert matches.** Many special-purpose game playing systems use a set of expert matches to train their evaluation functions. For many popular games, huge sets of expert matches are available; unfortunately, this is not true in the context of GGP, where nothing but the game rules is available. One way to overcome this problem would be to generate these “expert matches” oneself, for example by letting Fluxplayer play a series of games against itself, using its current fuzzy-logic based evaluation function and game-tree search. The time limit for the search would have to be in the same order of magnitude as that used during actual game play, since it is important that the “expert matches” are similar in quality to those encountered during actual game-play. One drawback of this method is that the generation of these matches would take much longer than the presented abstraction-based method.

If such “expert matches” were available, one could use the correlation between a feature and the actual game outcome as a feature selection criterion<sup>1</sup>. This would probably be a much more precise, albeit much more expensive feature selection method than the current abstraction-graph based one. Such a selection method should also solve the problem that in some games, good features were discarded as being “too special” since they matched too few states: In the “expert matches”, good features should match more often than in random play.

### 7.1.3 Evaluation Function Learning

Some games were difficult enough that the learner never found a solution during the initial random exploration. As a result, the learner always received the same reinforcement signal, which prevented further progress. If expert games were available, this problem could be addressed by bootstrapping the learning process from these games and continue learning using the implemented TD( $\lambda$ ) method.

On several games, the Fluxplayer opponent was either too weak, too strong, or – in one case – behaved in a way that blocked progress. This effect could be countered by using self-play: playing against a backed-up version of the learner with negated weights, and performing a new backup whenever the winning ratio has reached a given threshold. In general, self-play is shown to produce good results, but takes longer to converge than playing against a competent opponent.

The time needed for the evaluation function could be reduced by successively pruning features that constantly have a near-zero weight.

---

<sup>1</sup>This criterion is not infallible, since it is very difficult to determine a priori the quality of any predictor value in a multiple regression model, but it should be good enough as a selection criterion.

To counter the problem the learner had with some of the simultaneous games, one could look into alternatives to the current “randomize all opponents” approach. A careful evaluation would have to be done if the greater precision of such a method is worth the associated cost and the resulting lower number of training matches.

## 7.2 Summary

In this thesis, Fawcett’s ideas have been applied to the context of GGP for the first time. With GDL, a language was used that provides less explicit information about a game than Zenith’s Prolog domain theory. In some cases, this required a more elaborate approach than Zenith’s feature transformations, for example the automatic deduction of operator pre-images for goal regression.

The combination of Zenith’s feature transformations with reinforcement learning was explored. This required to use a non-iterative approach to feature generation and selection.

A feature selection method had to be devised that can operate without feedback from learning, and without any extra information such as expert matches. With abstraction-graph-based feature selection, a new approach that meets these requirements was developed. It is based on the explicit analysis of abstraction relationships between the features and allows to deduce the degree of abstraction for any feature analytically.

The complete system has been evaluated on an extensive set of games to ensure that the system’s parameters were not biased towards any particular game. With Fluxplayer’s fuzzy-logic-based evaluation function, a state-of-the-art general game player has been chosen as the baseline opponent. The system successfully generated an evaluation function that outperformed its opponent in several games.



## Appendix A Overview of the Games

**Asteroids** A single-player game where the player controls a spaceship on a  $20 \times 20$  board. The primary goal is to make the ship stop at a given position; the secondary goal is to stop the ship anywhere.

**Beatmania** A conversion of a video game to GGP; originally a single-player game, in this version one player (the *dropper*) takes the role of the video machine, dropping blocks in one of three slots. These fall down (Tetris-like) and must be caught by the *player* at the bottom. Since the *dropper* always receives 100 points, this game only makes sense for the *player* and could be classified as a single-player game.

**Blobwars** A two-player strategy game on an  $8 \times 8$  board. The players' "blobs" can jump two spaces or duplicate into an adjacent cell; the opponent's pieces can be converted. Whoever has the most blobs when the board is filled wins.

**Blocker** A game for two players, the blocker and the crosser, played on a  $4 \times 4$  board. Both players mark fields simultaneously. If both choose the same field, the blocker takes precedence. The crosser's goal is to form a bridge of connected fields, the blocker must prevent that.

**Bomberman** An arcade game for two players. Both can simultaneously move around and drop bombs. The goal is to blow up the opponent.

**Breakthrough** A two-player strategy game played on an  $8 \times 8$  board. Both players control 16 pieces that can move and capture. Whoever reaches the opponent's home row first wins.

**Checkers** Checkers, also known as Draughts, is a two-player strategy game on an  $8 \times 8$  board. Each player has 12 pieces that can move and capture forward.

**Chinese Checkers** A simplified version of the popular board game Chinese Checkers for three players; only three pieces each. Move all your pieces to the other side of the board by moving and jumping over other pieces.

**Circle Solitaire** A single-player game involving eight places in a circle that hold either a green or red disk or are empty. The player can remove any green disk, causing its place to be empty and the disks on the two adjacent places to be flipped, if there are any. Another option is to reset the game to the original state. The goal is to remove all disks with as few resets as possible.

**Connect-Four** A two-player game where the players drop coins into one of seven slots and try to make a line of four pieces.

**Crisscross** A smaller version of Chinese Checkers for two players, with four pieces each on a rectangular board.

**Crossers3** A three-player game with simultaneous moves, played on a triangular board. Each player starts in a corner and has to reach the opposite edge. While moving over the board, the pieces leave a trail of walls, so it is possible to get stuck. The earlier a player reaches its goal position, and the fewer opponents do, the higher each player's reward.

**Eight-Puzzle** A sliding puzzle where the player's goal is to move eight numbered pieces into order.

**Endgame** A Chess endgame position. White king and white rook against black king.

**Ghostmaze** A game with two asymmetric roles, the ghost and the explorer. The explorer has to find the exit by moving through a maze of walls; the ghost can move through any wall and drop slime. The ghost has won when the explorer walks into a cell that is occupied by slime, or the ghost drops slime while in the same cell as the explorer.

**Hallway** A two-player game played on a  $9 \times 9$  board. Each player controls a piece that must reach the opposite row. Also, walls can be placed onto the board to restrict movement.

**Hanoi** The single-player game Tower of Hanoi. Five discs and three pillars. Find a way to move all discs to the third pillar while never placing a bigger disc on a smaller one.

**Incredible** A composite single-player game: The well-known blocks world with six blocks and a simple single-player game called maze.

**Knightmove** A puzzle also known as the Knight's Tour. The player controls a single piece that moves according to the regular rules for a knight in chess. It starts on an empty chess board and must visit each state exactly once.

**Merrills** An abstract strategy two-player game played on an irregular board, also known as Nine Men's Morris. Each player has nine pieces (called men) that can move between the board's twenty-four intersections. When a connected line of three pieces is formed, it is called a "mill" and allows the player to remove one of the opponent's men. The last player to have three or more men wins.

---

**Minichess** An endgame position from a smaller version of Chess on a  $4 \times 4$  board. White rook and white king against black king. White has to mate in 10 moves.

**Mummy Maze** Another two-player maze game, where the explorer has to find the exit and the mummy has to catch him. The explorer starts nearer to the exit, but the mummy can move twice as fast.

**Othello** The two-player strategy board game Othello, also known as Reversi. Players take turns in placing disks on the board; all disks of the opponent that are caught in a “span” between two own disks by a move are converted to the moving player’s color. The player with the most disks in the end wins.

**Pacman** The famous arcade game for three players: Pacman and two ghosts. Pacman has eat as many pellets as possible, while the ghosts try to catch him.

**Pancakes** A single-player puzzle. Flip eight pancakes into the right order.

**Peg** Peg Solitaire is a single-player board game. The board is initially filled with pegs except for the central hole. The goal is to remove all pegs except one in the central position.

**Pentago** Pentago is a two-player game played on a  $6 \times 6$  board that is divided into four  $3 \times 3$  quadrants. A move consists of placing a piece on the board and then rotating one of the quadrants by 90 degrees. The first player to form a line of five pieces wins.

**Quarto** Quarto is a two-player game played on a  $4 \times 4$  board. The 16 pieces have four dichotomous attributes each: color, shape, height and consistency. The piece that is to be played next is chosen by one of the players, then the opponent has to place it. The goal is to make a row of four pieces that conform in one attribute.

**Racetrack Corridor** A race between two players, both moving simultaneously in separate lanes. Also, walls can be placed to slow down the opponent.

**Skirmish** A chess variant on a regular chess board, but only one king, one rook, two knights and four pawns per side. Contrary to regular chess rules, there is no check; the king acts as a regular piece. The goal of the game is to capture as many of the opponent’s pieces as possible.

**Tic-Tac-Toe** The popular children’s game Tic-Tac-Toe, or Noughts and Crosses. Place X and O marks and get three in a line.

**Tic-Tic-Toe** Simultaneous Tic-Tac-Toe.

**Wallmaze** A two-player game with simultaneous moves, where the two players have to exchange places in a maze. There is a reward for reaching one's goal position, with extra points for being the first or even only one to do so. The two players can block each other.



## **Appendix B Training graphs**

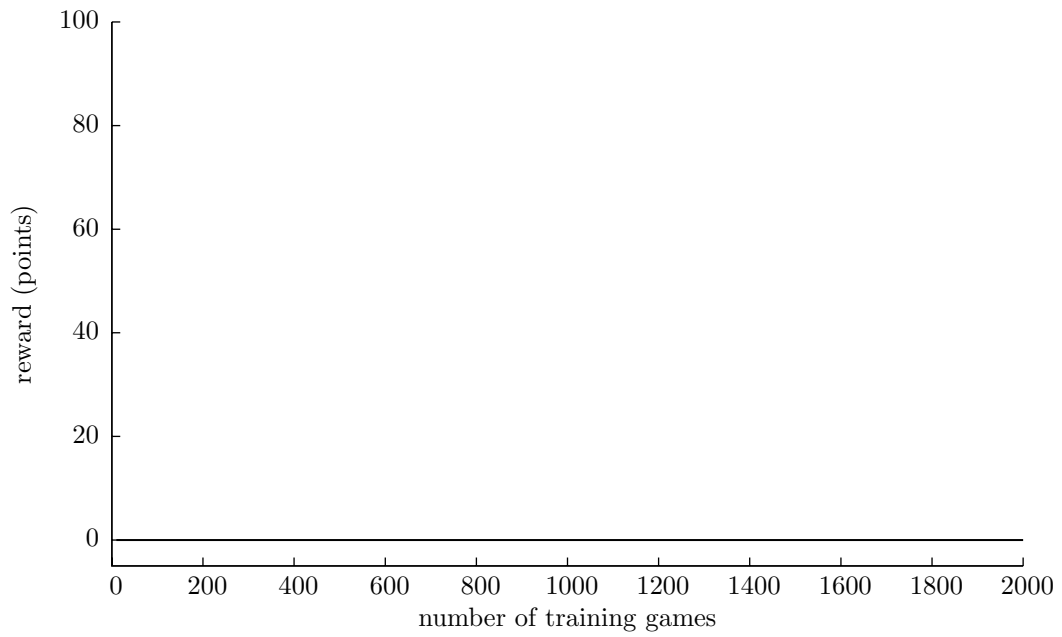


Figure B.1: Training rewards for Eight-Puzzle (player)

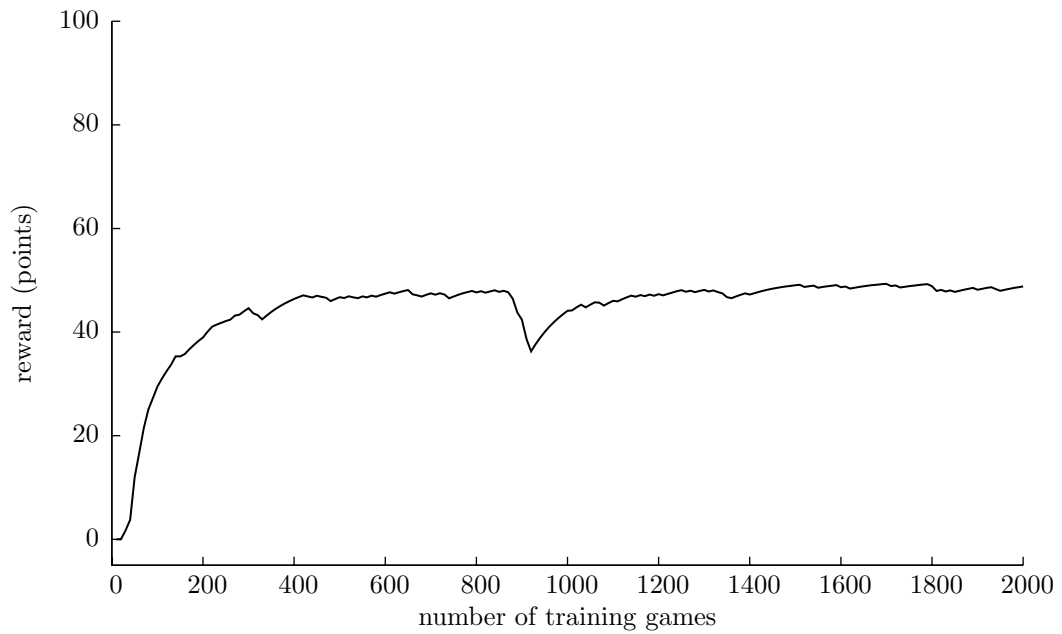


Figure B.2: Training rewards for Asteroids (ship)

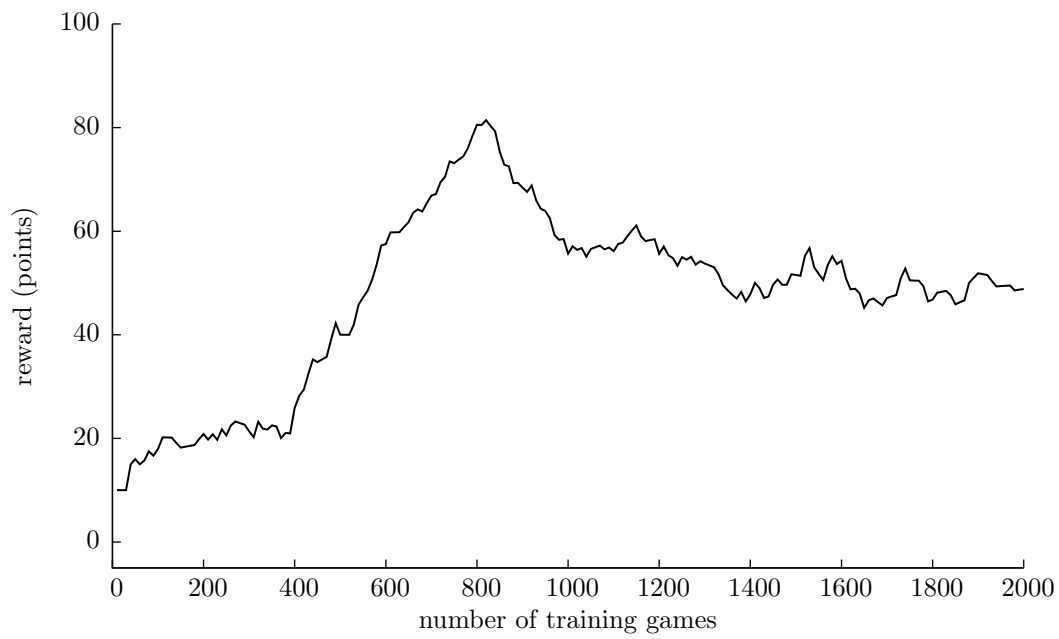


Figure B.3: Training rewards for Blocker (blocker)

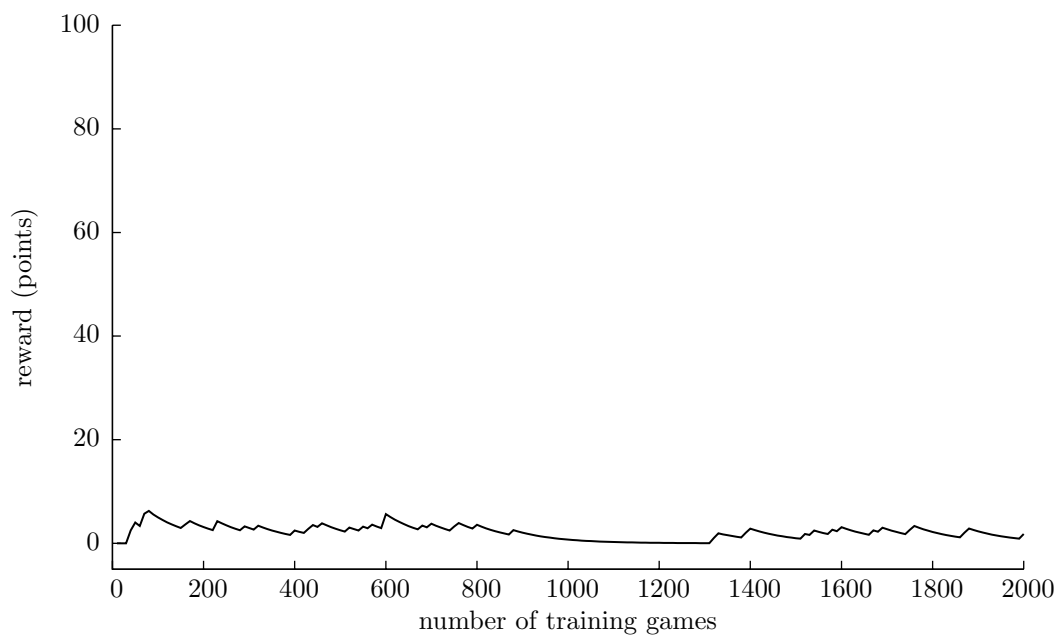


Figure B.4: Training rewards for Blocker (crosser)

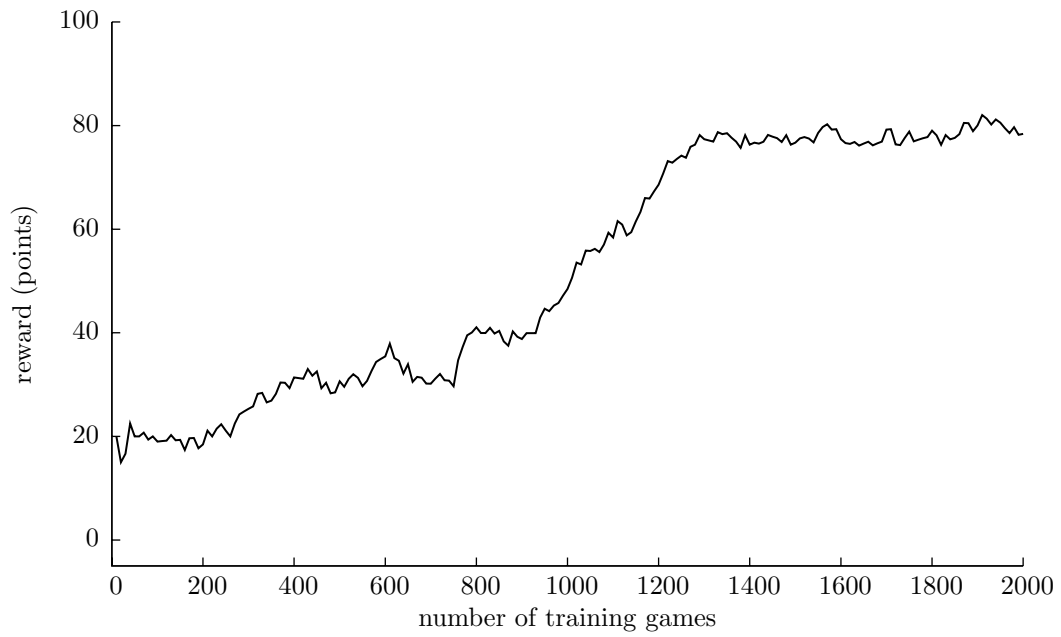


Figure B.5: Training rewards for Bomberman (bomberman)

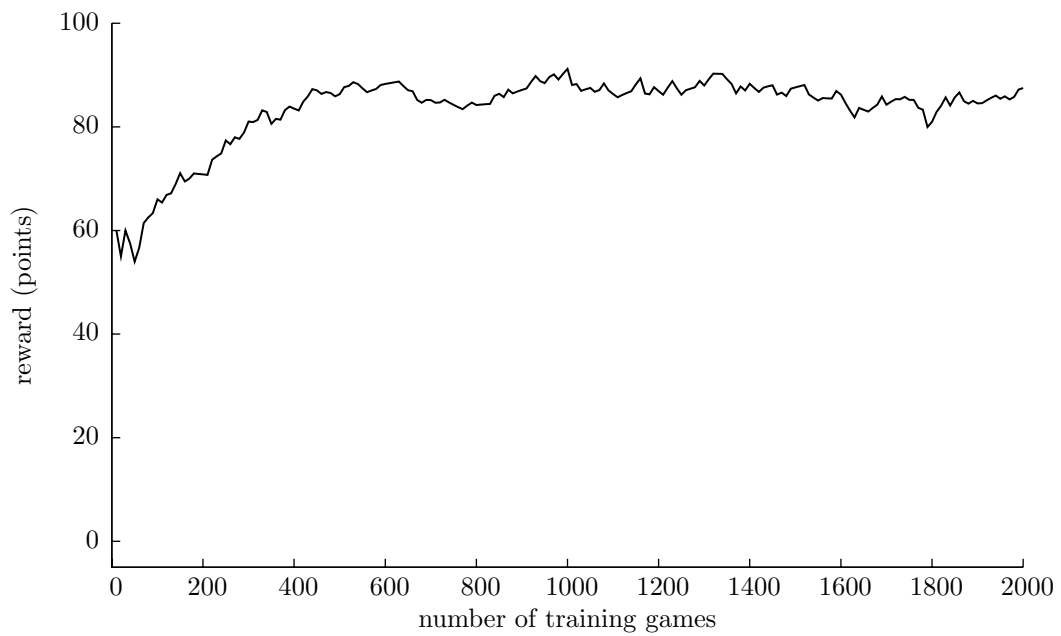


Figure B.6: Training rewards for Bomberman (bomberwoman)

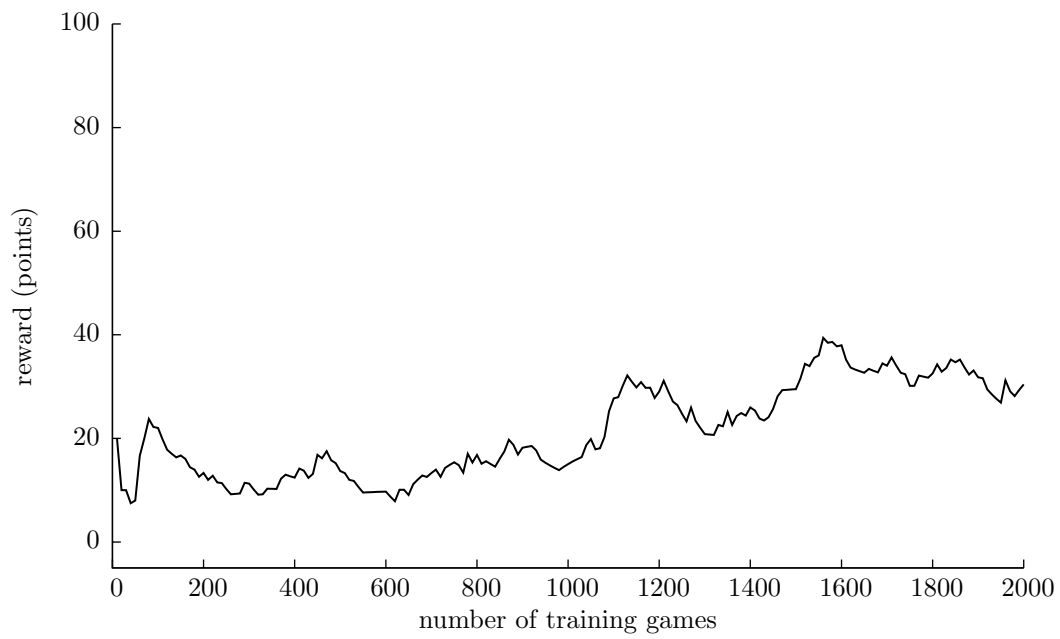


Figure B.7: Training rewards for Breakthrough (black)

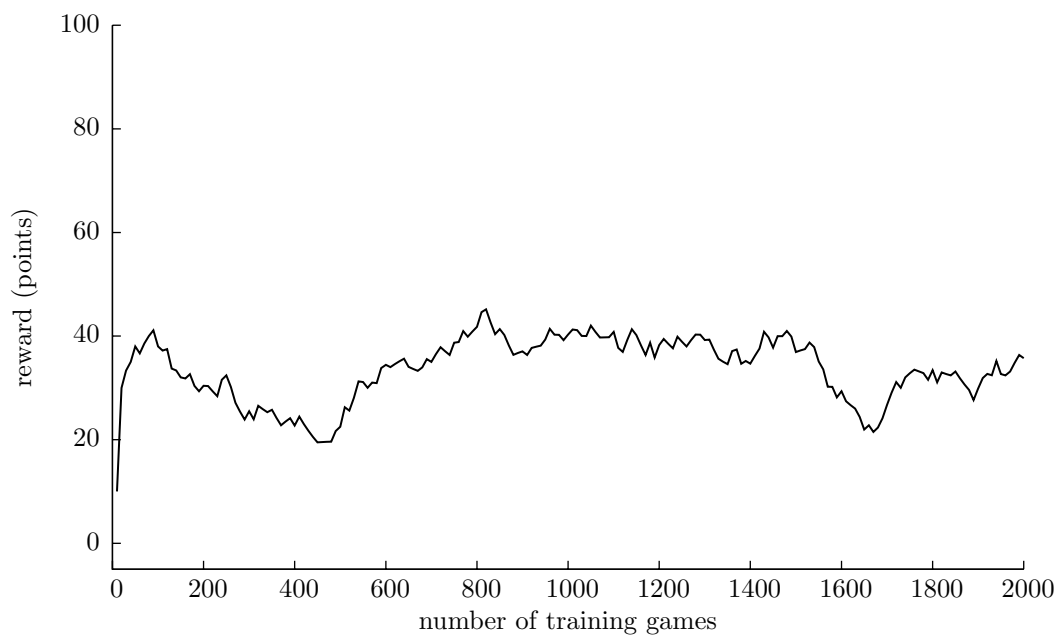


Figure B.8: Training rewards for Breakthrough (white)

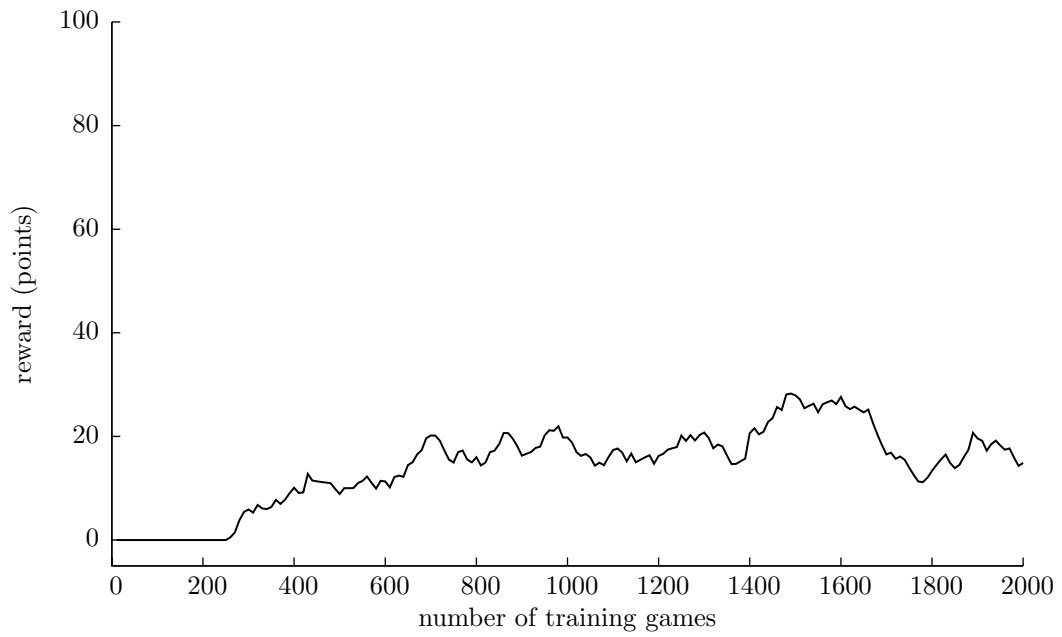


Figure B.9: Training rewards for Checkers (black)

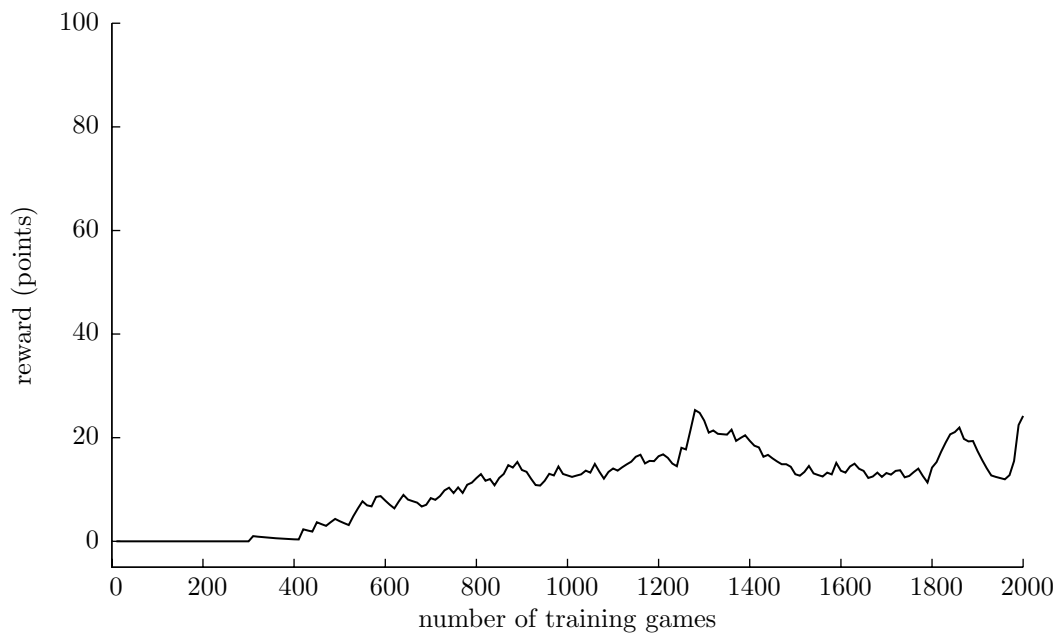


Figure B.10: Training rewards for Checkers (white)

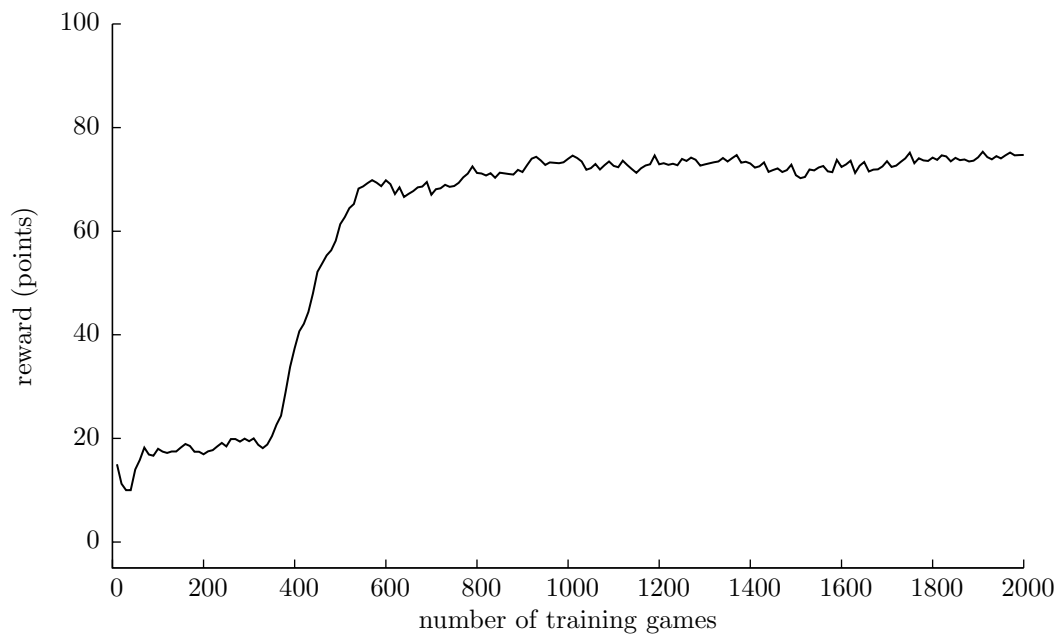


Figure B.11: Training rewards for Chinese Checkers (blue)

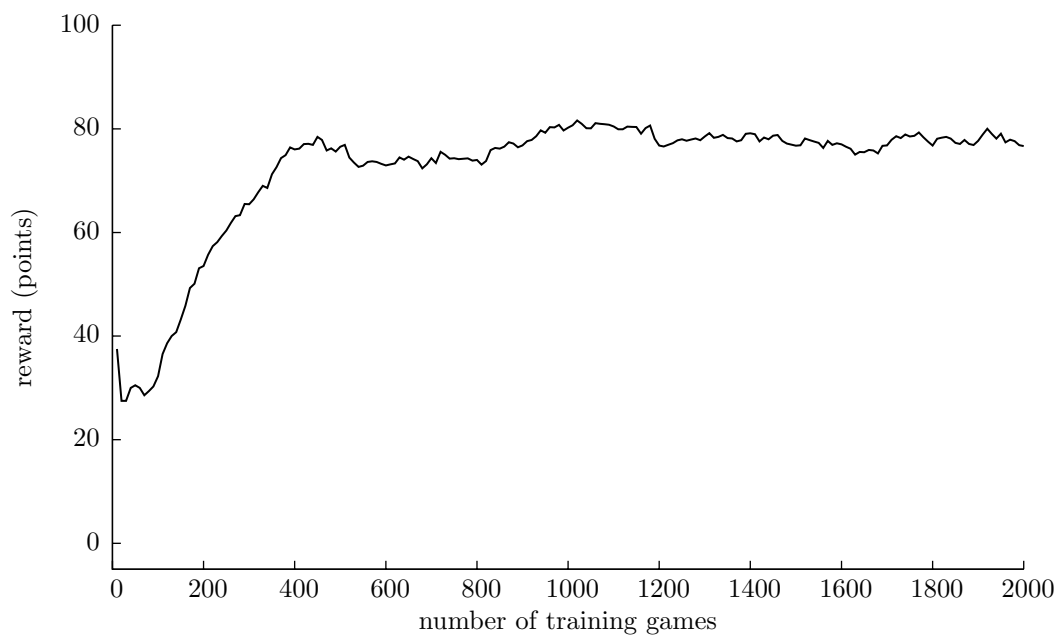


Figure B.12: Training rewards for Chinese Checkers (green)

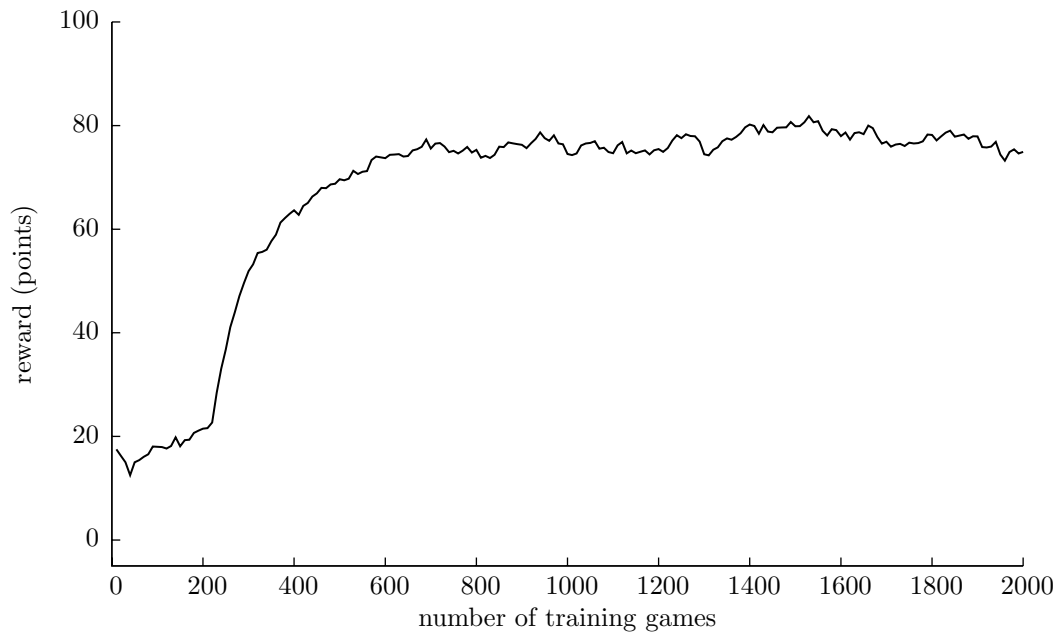


Figure B.13: Training rewards for Chinese Checkers (red)

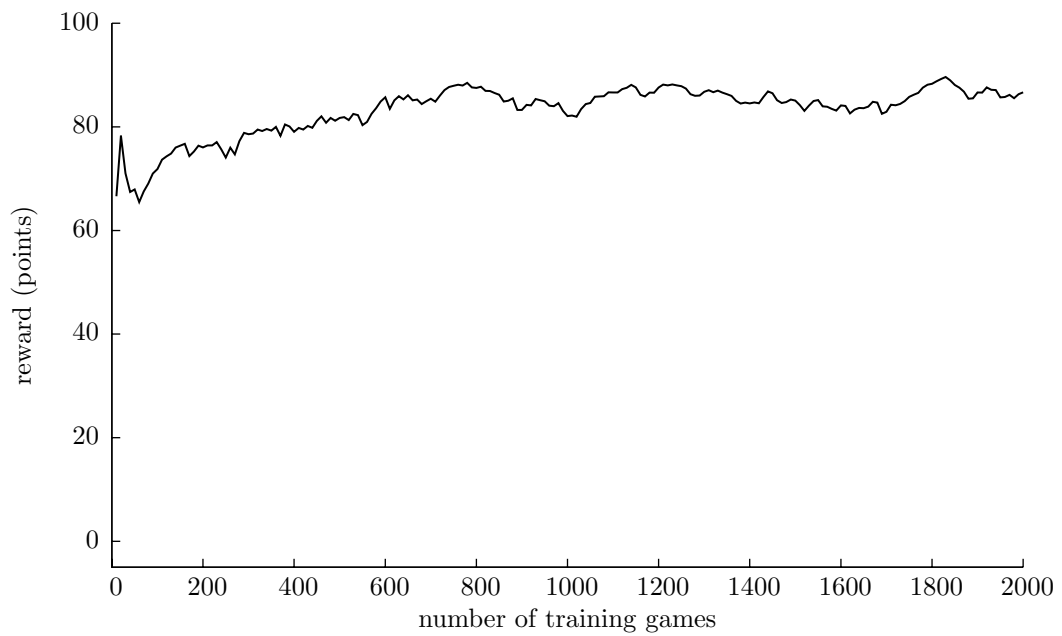


Figure B.14: Training rewards for Circle Solitaire (taker)



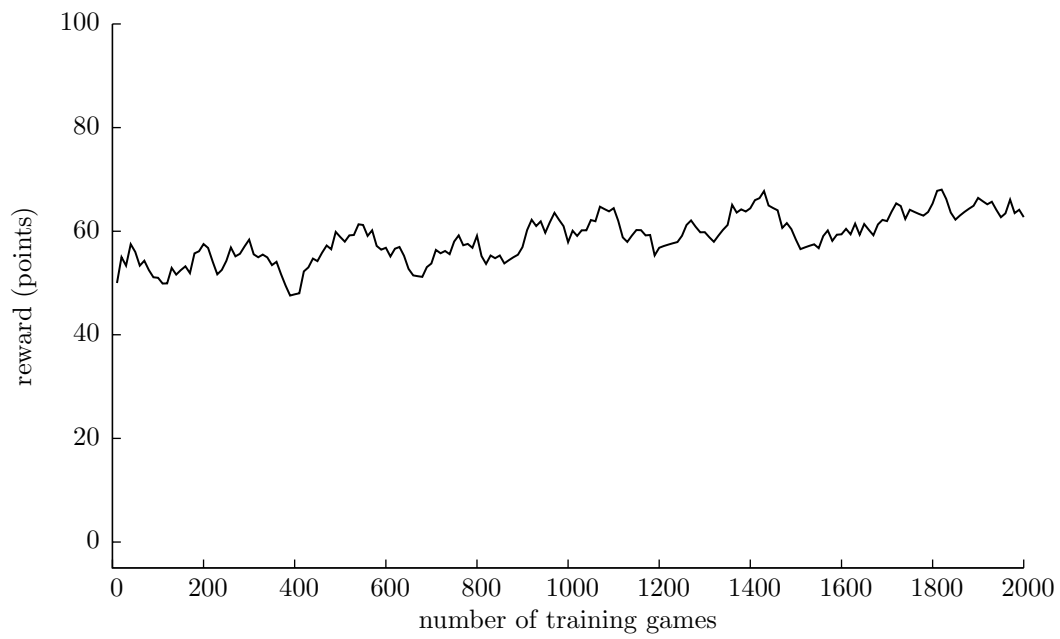


Figure B.15: Training rewards for Connect-Four (red)

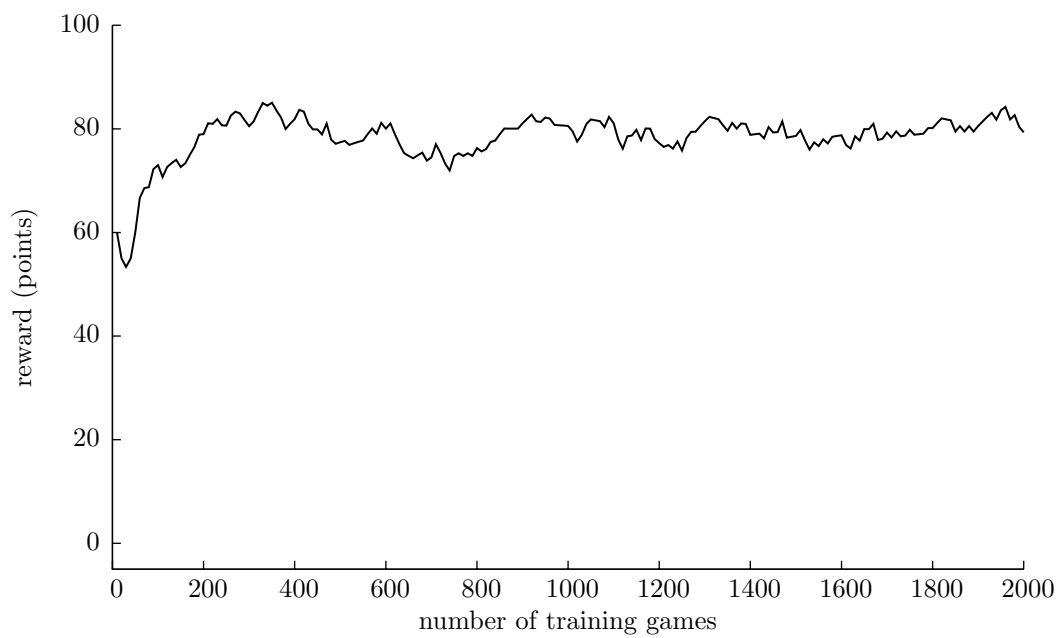


Figure B.16: Training rewards for Connect-Four (white)

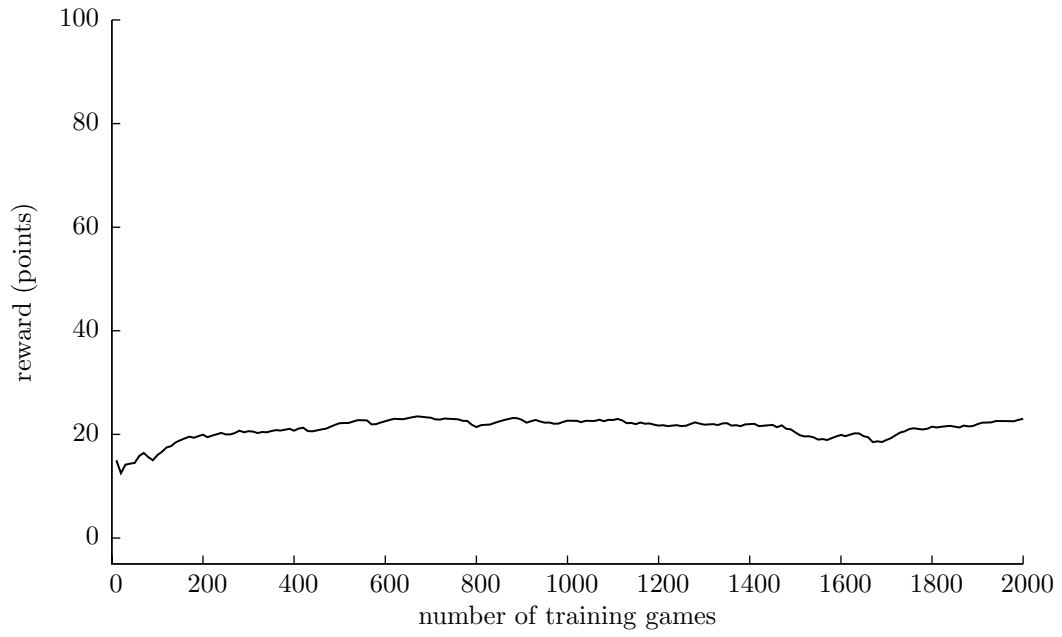


Figure B.17: Training rewards for Crisscross (red)

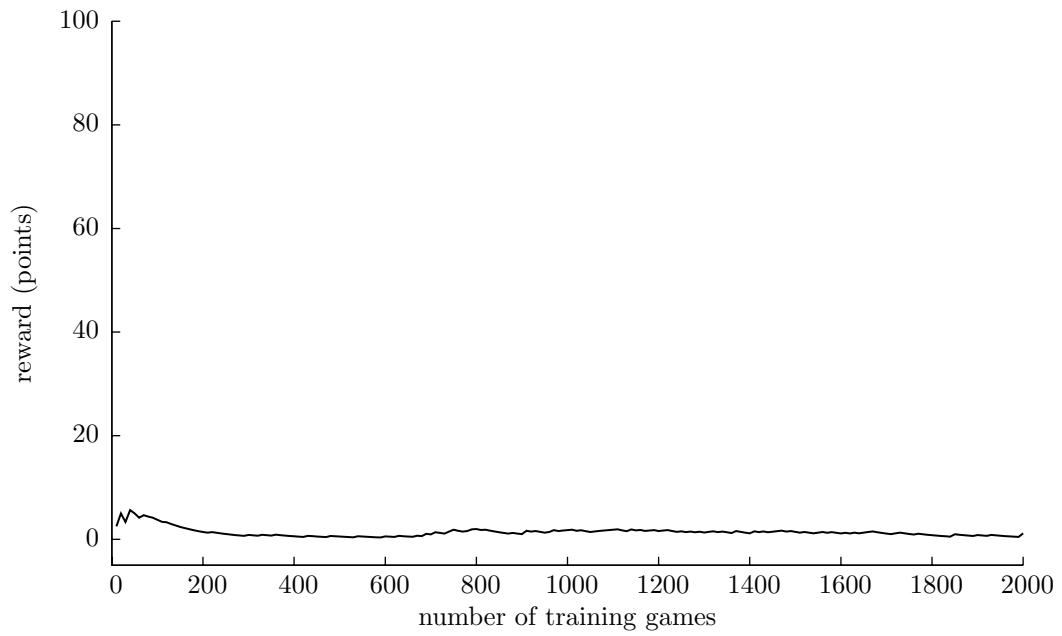


Figure B.18: Training rewards for Crisscross (teal)

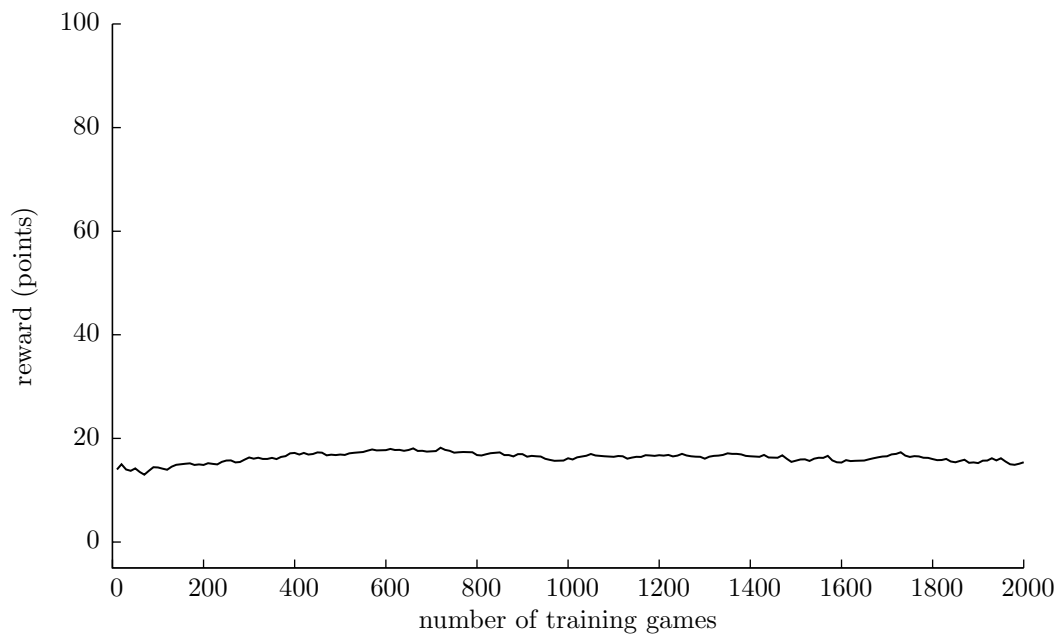


Figure B.19: Training rewards for Crossers3 (left)

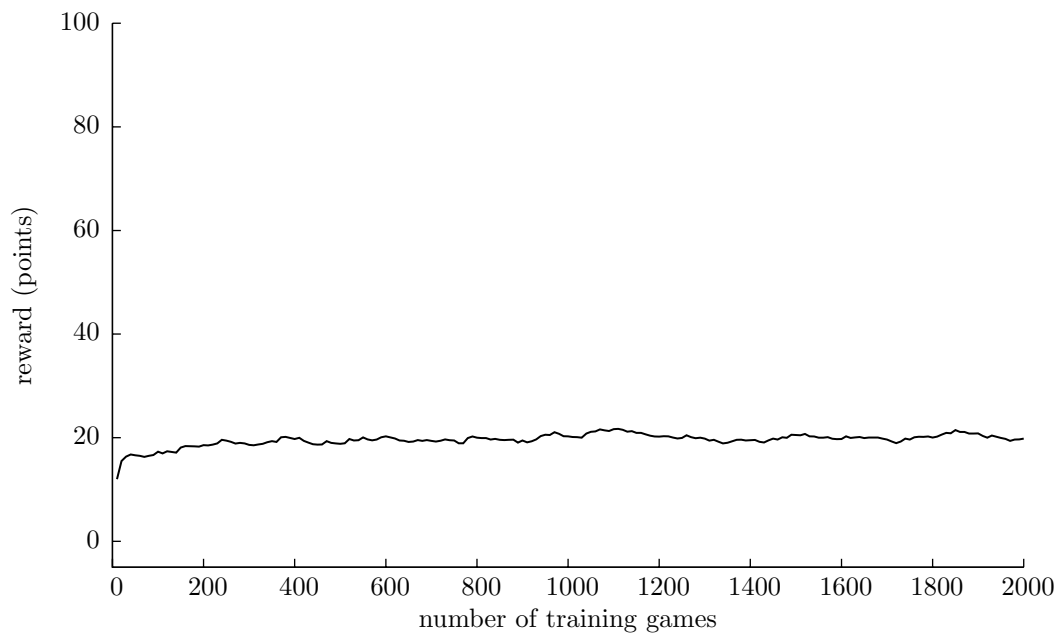


Figure B.20: Training rewards for Crossers3 (right)

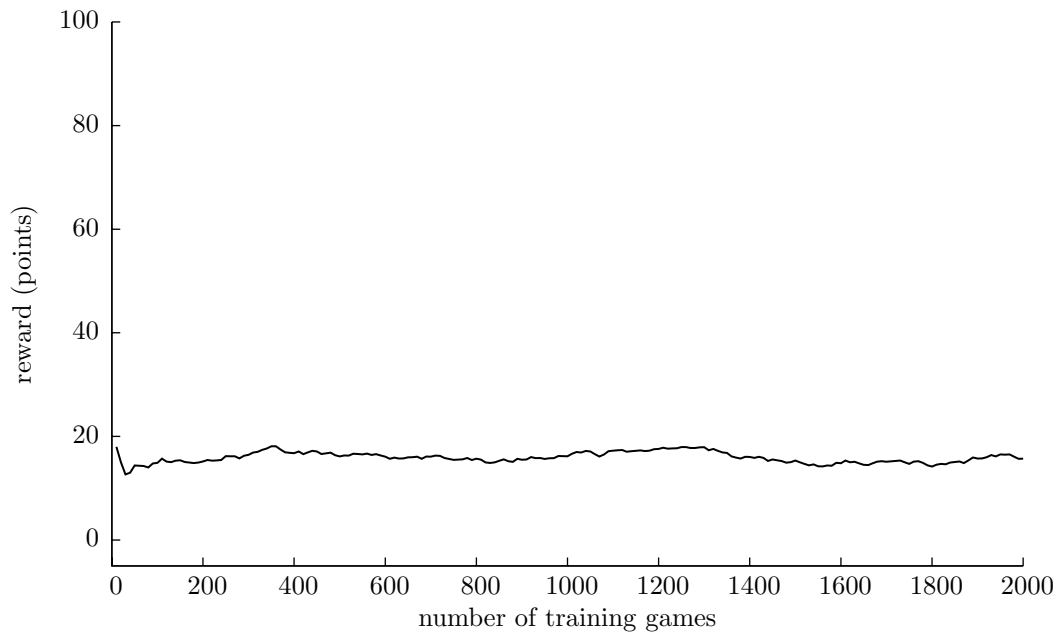


Figure B.21: Training rewards for Crossers3 (top)

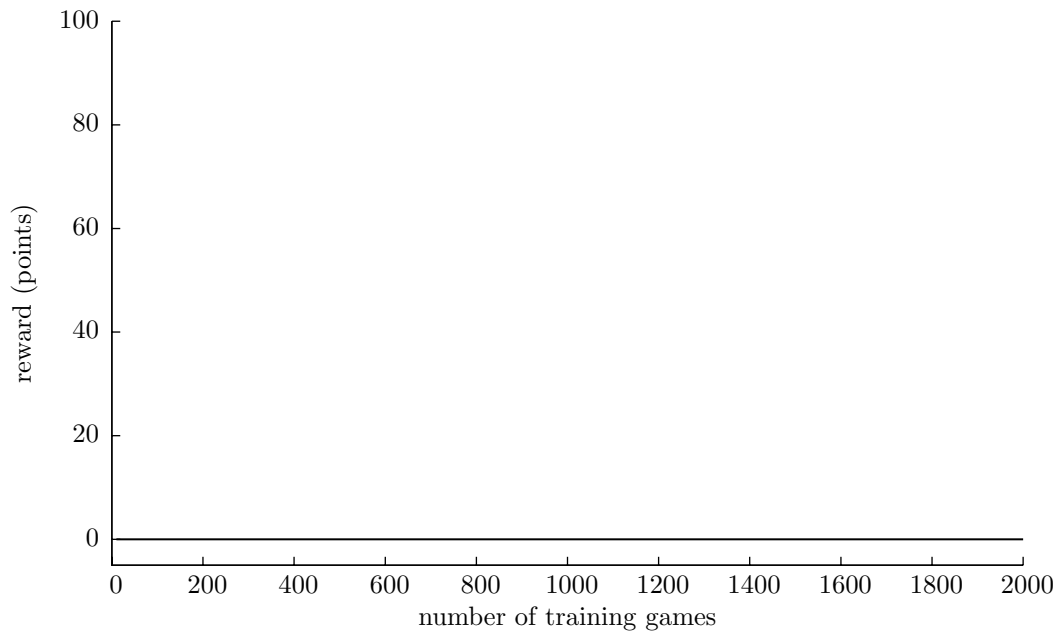


Figure B.22: Training rewards for Incredible (robot)

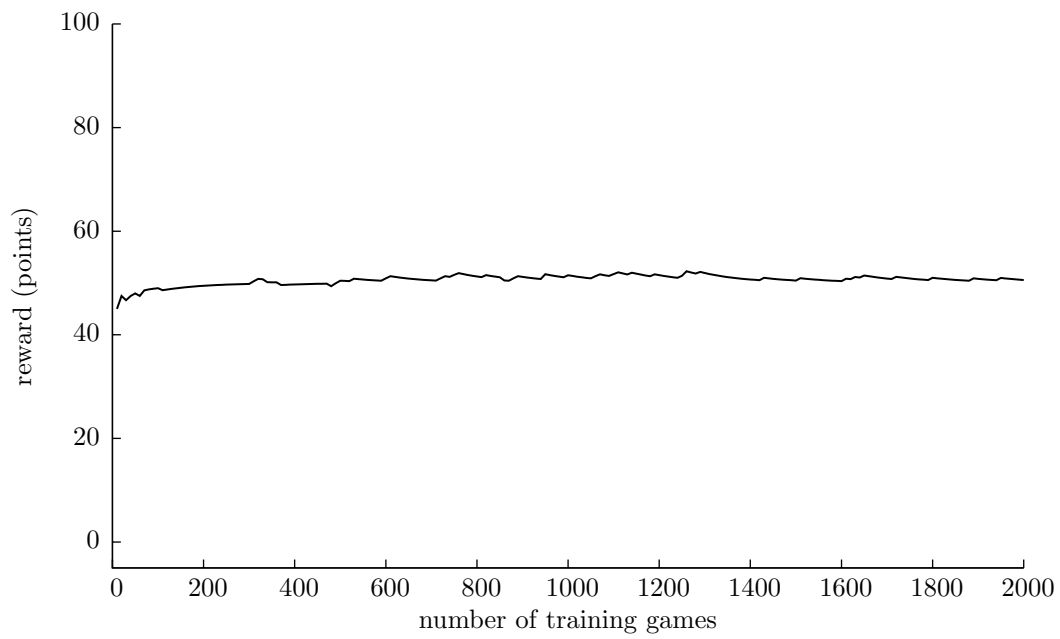


Figure B.23: Training rewards for Ghostmaze (explorer)

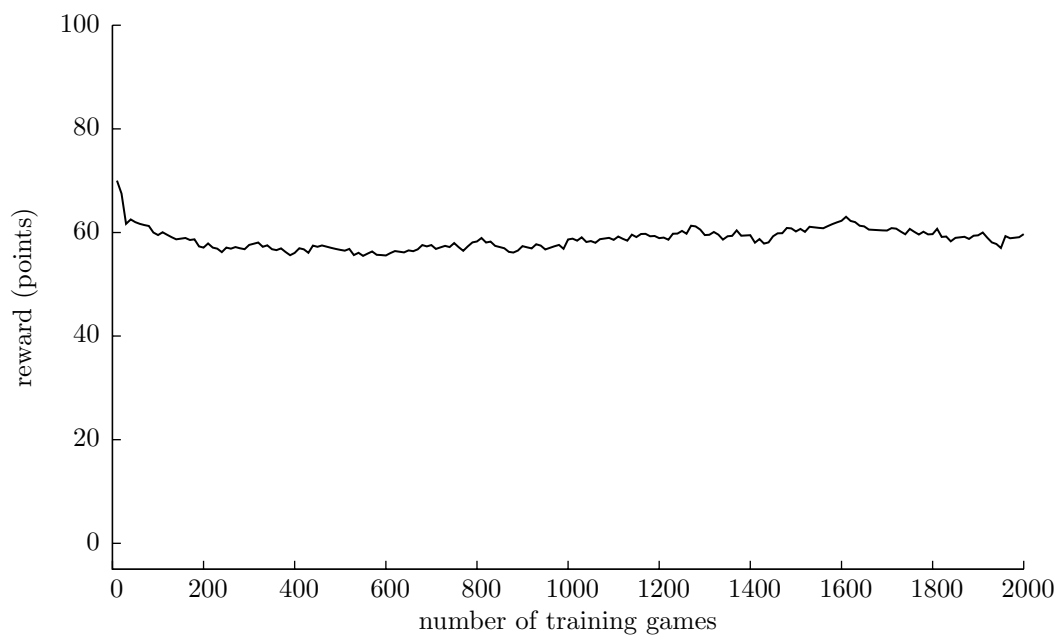


Figure B.24: Training rewards for Ghostmaze (ghost)

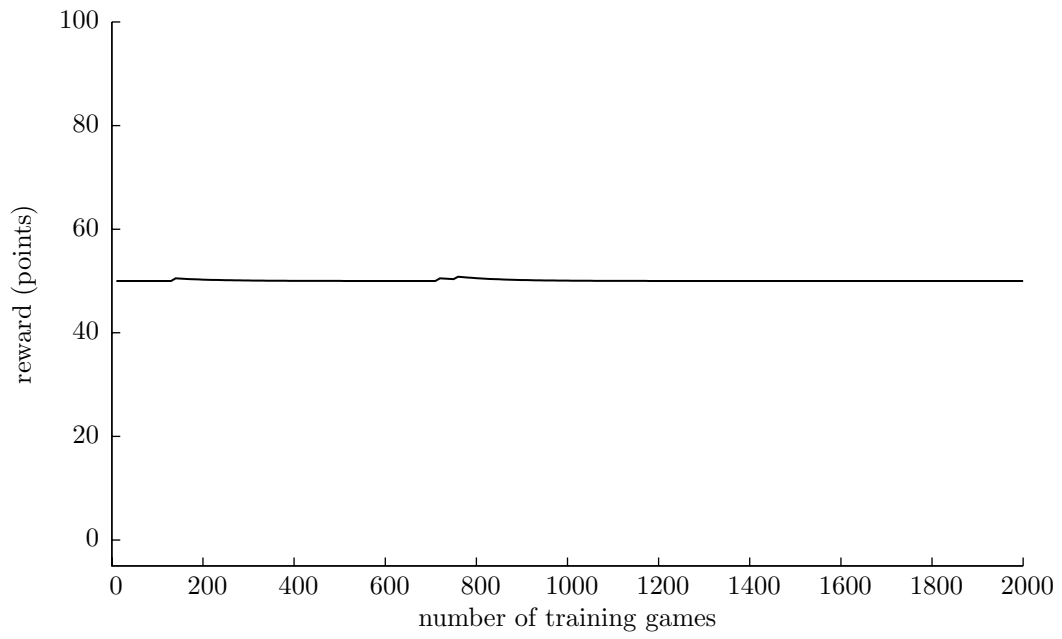


Figure B.25: Training rewards for Hallway (black)

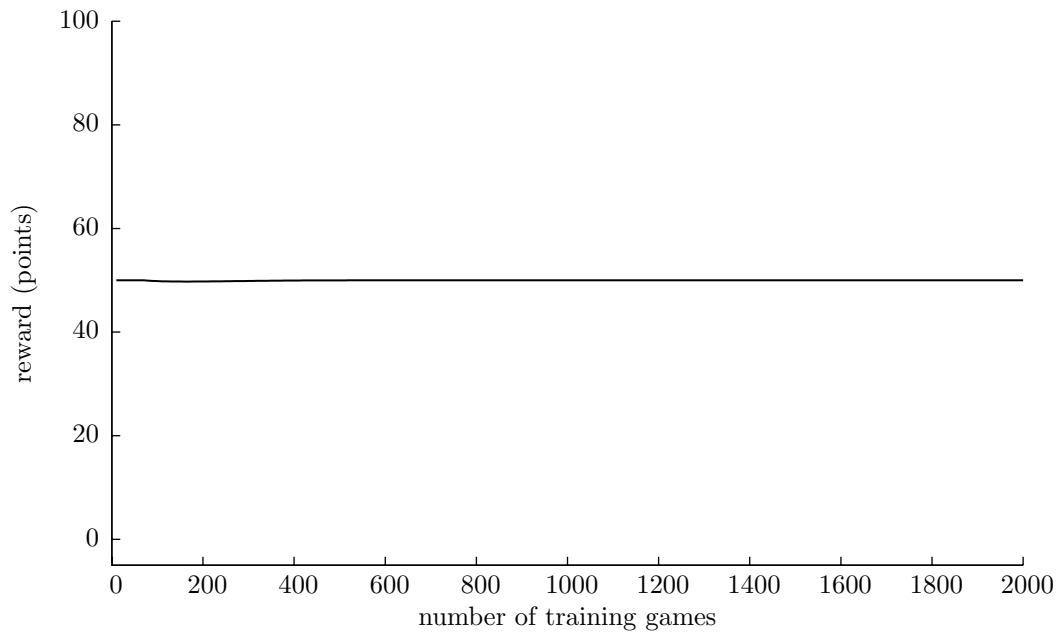


Figure B.26: Training rewards for Hallway (white)

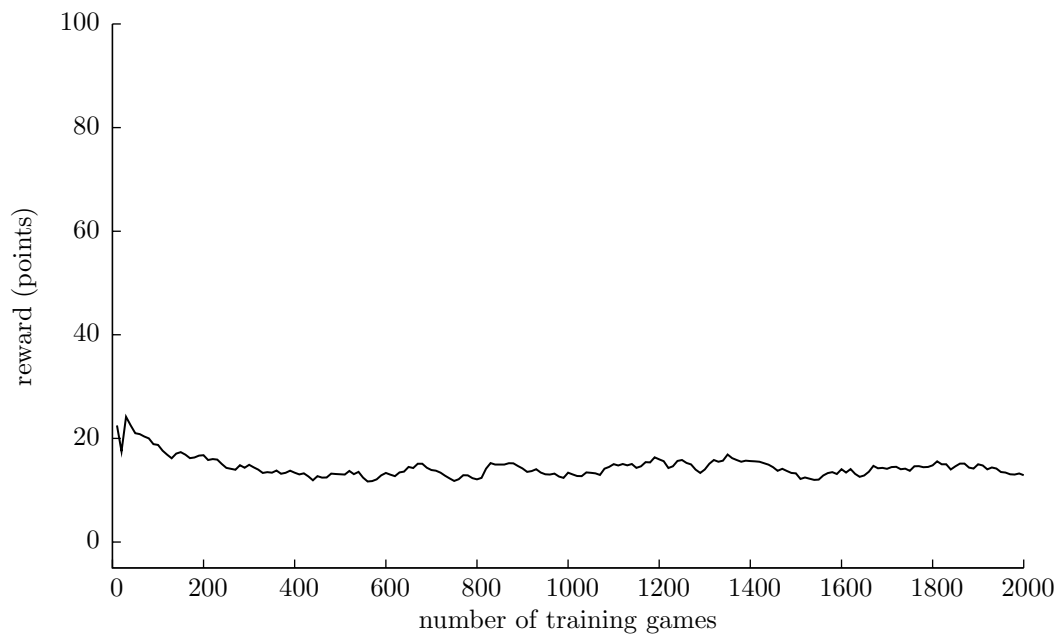


Figure B.27: Training rewards for Merrills (black)

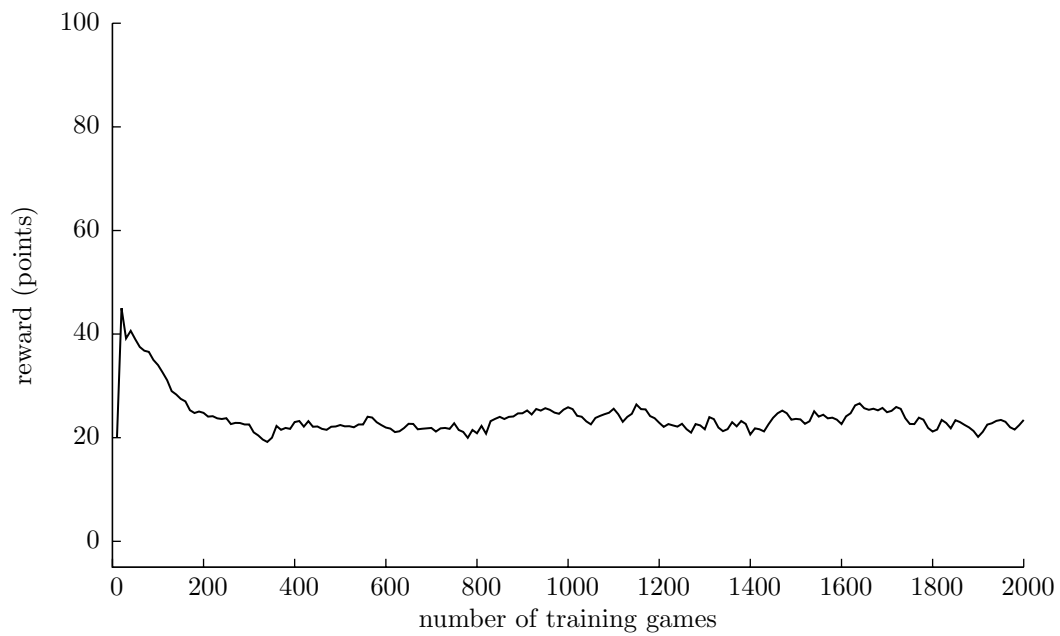


Figure B.28: Training rewards for Merrills (white)

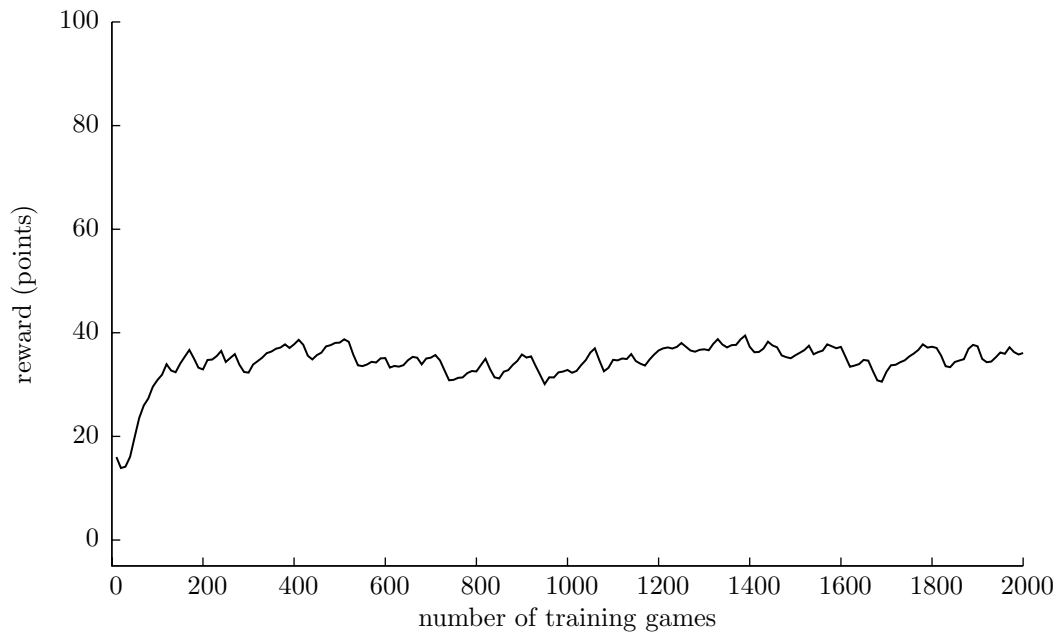


Figure B.29: Training rewards for Pacman (pacman)

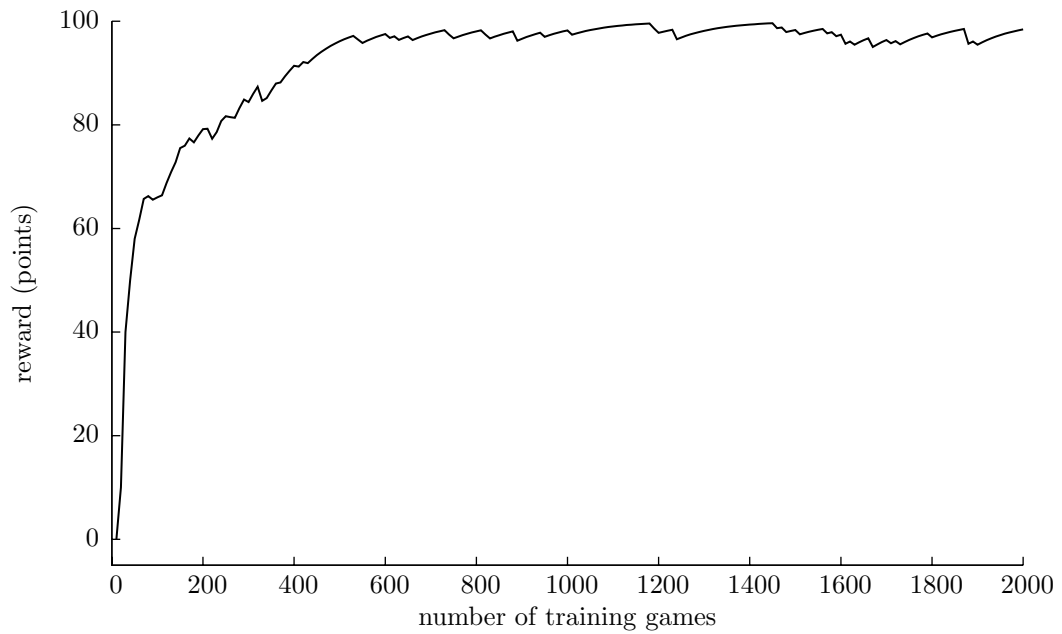


Figure B.30: Training rewards for Pacman (inky)



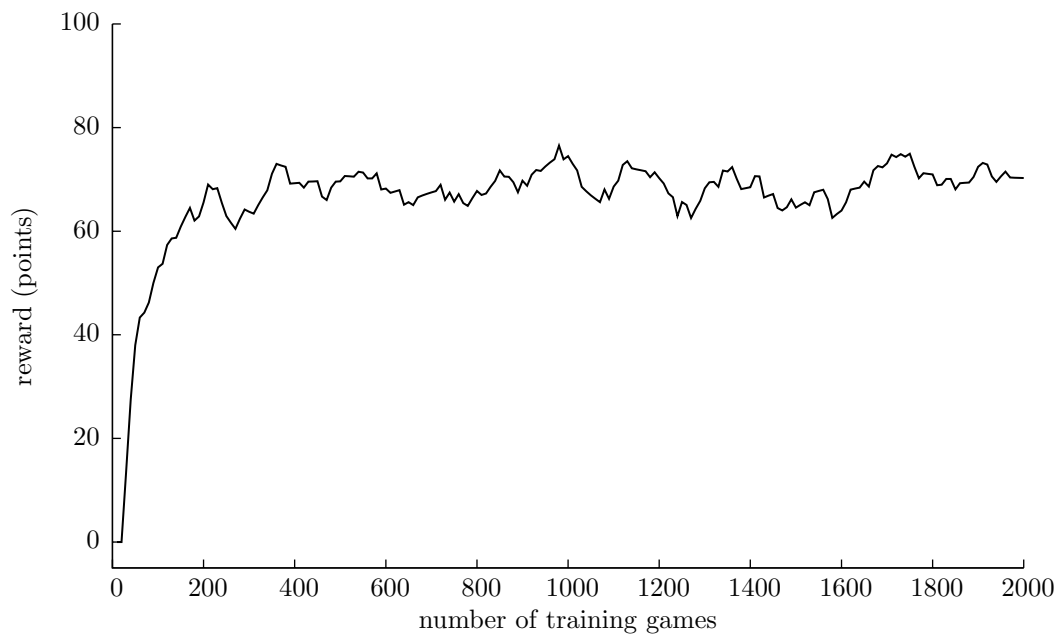


Figure B.31: Training rewards for Pacman (blinky)

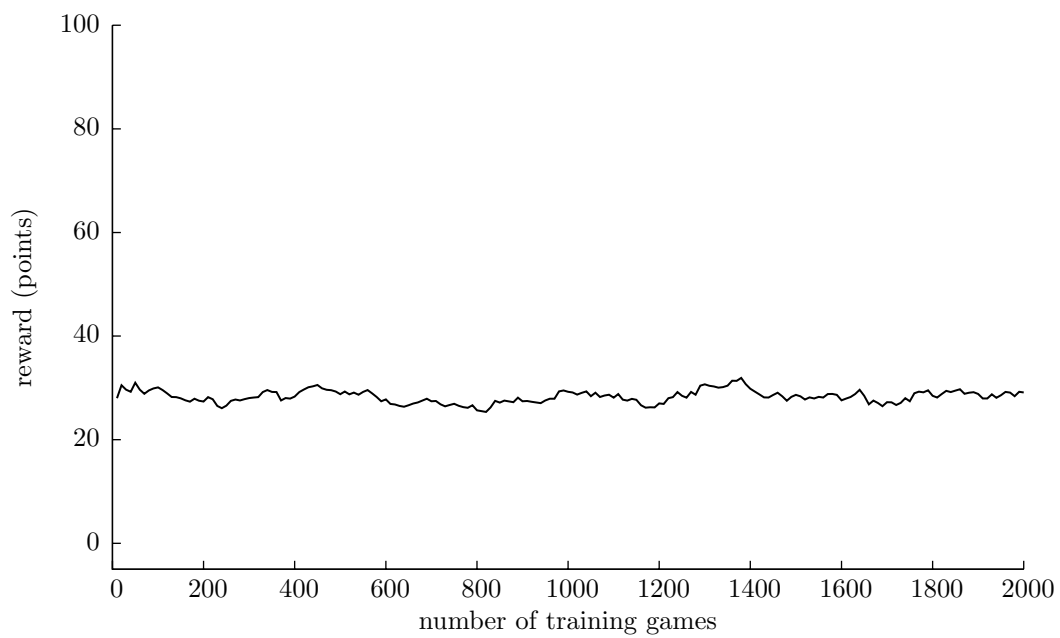


Figure B.32: Training rewards for Peg (jumper)

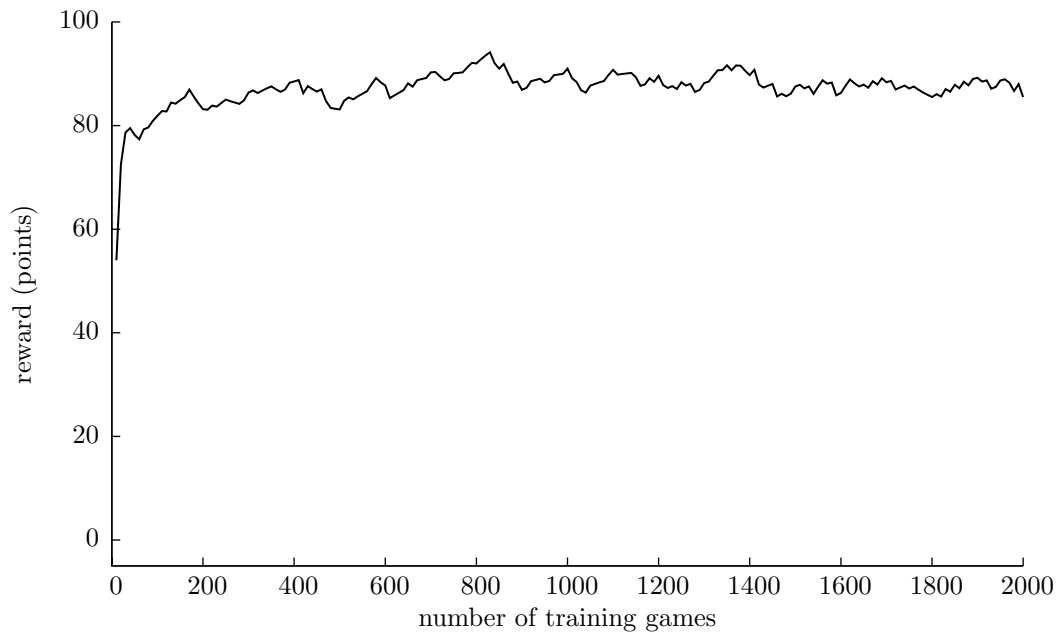


Figure B.33: Training rewards for Racetrack Corridor (black)

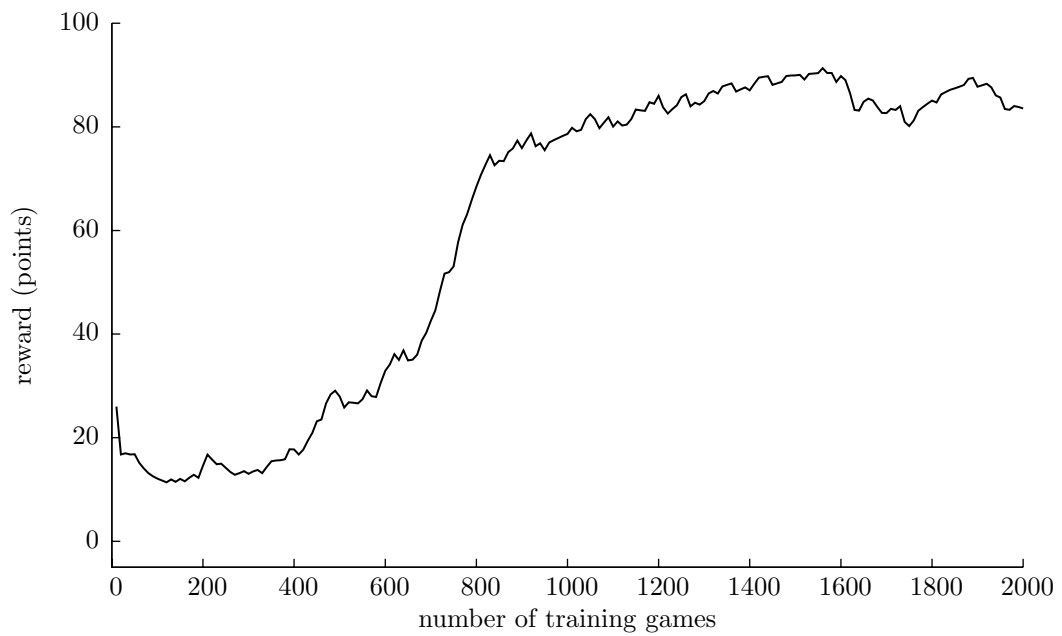


Figure B.34: Training rewards for Racetrack Corridor (white)

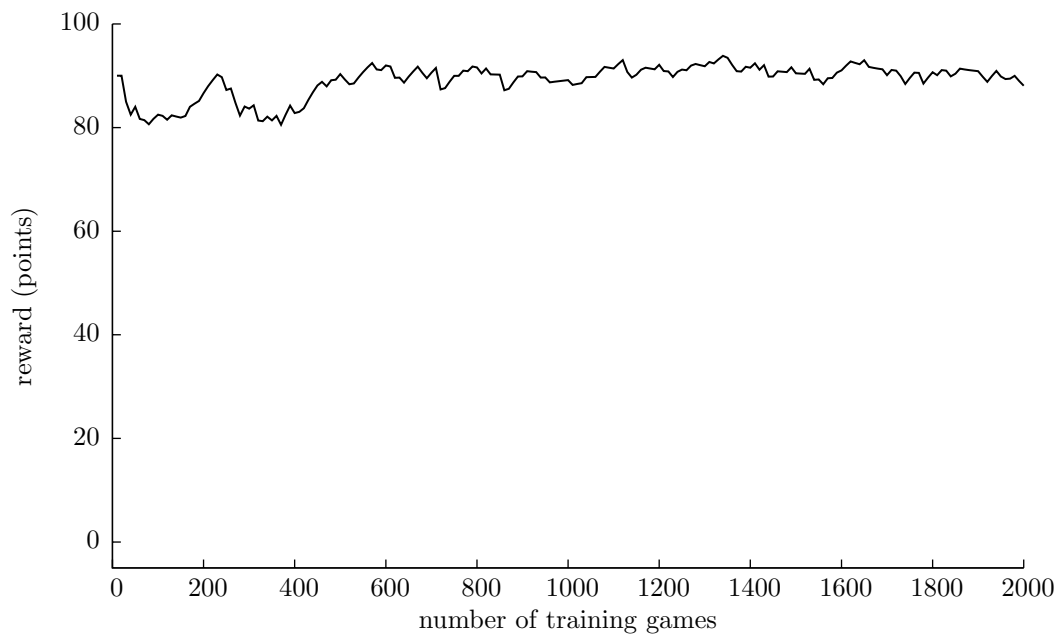


Figure B.35: Training rewards for Tic-Tac-Toe (xplayer)

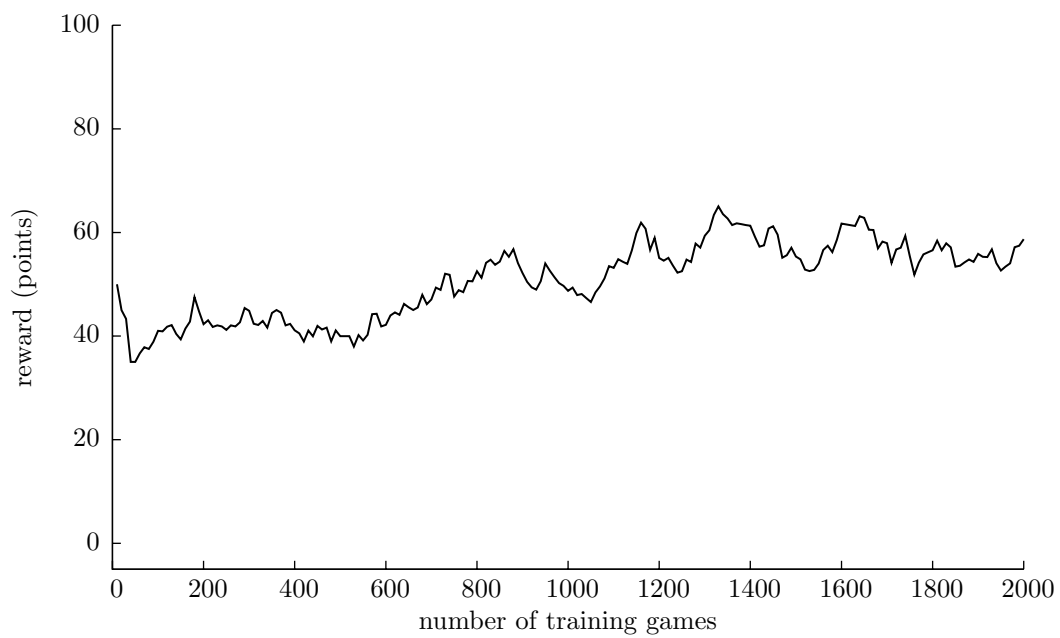


Figure B.36: Training rewards for Tic-Tac-Toe (oplayer)

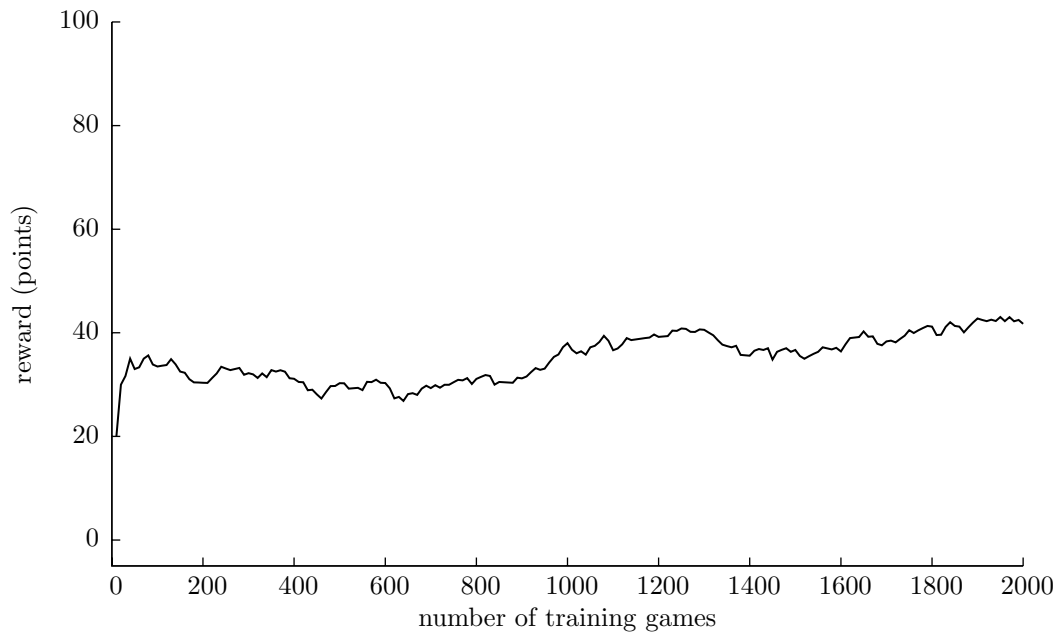


Figure B.37: Training rewards for Tic-Tac-Toe (black)

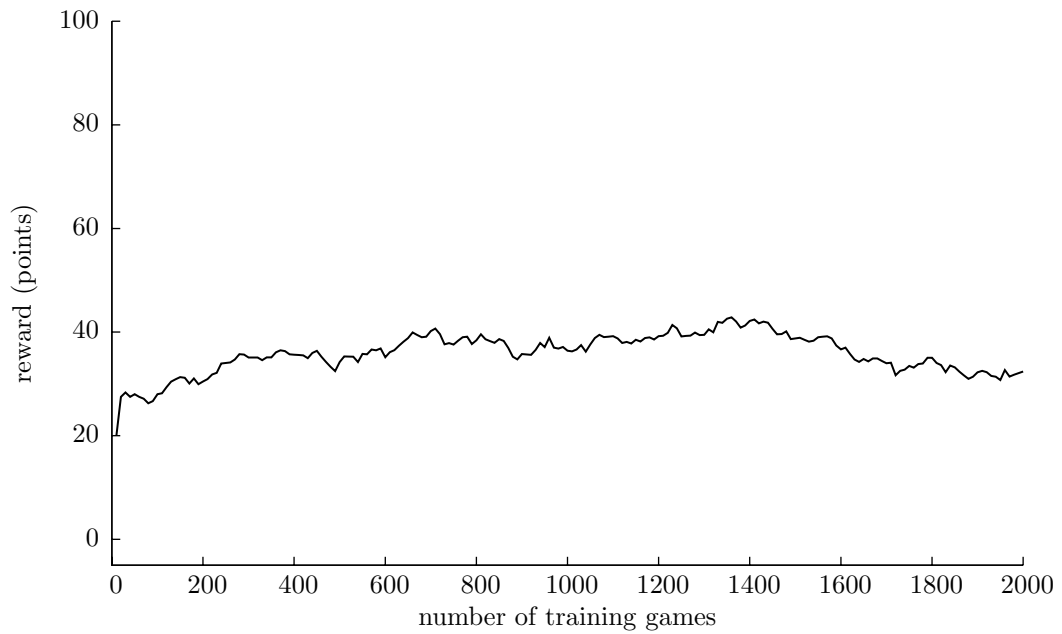


Figure B.38: Training rewards for Tic-Tac-Toe (white)

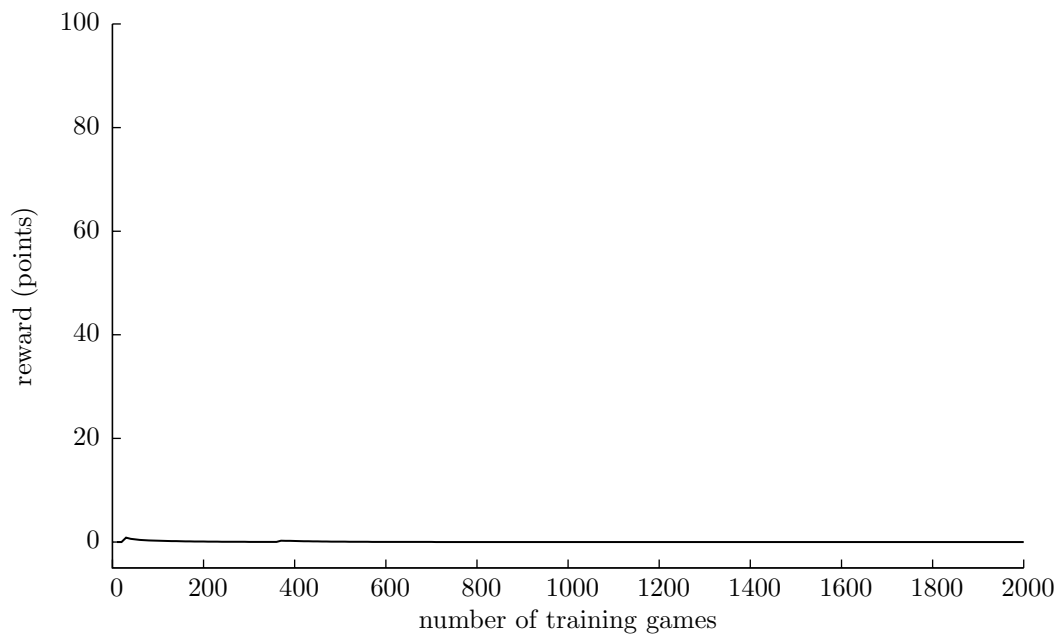


Figure B.39: Training rewards for Wallmaze (black)

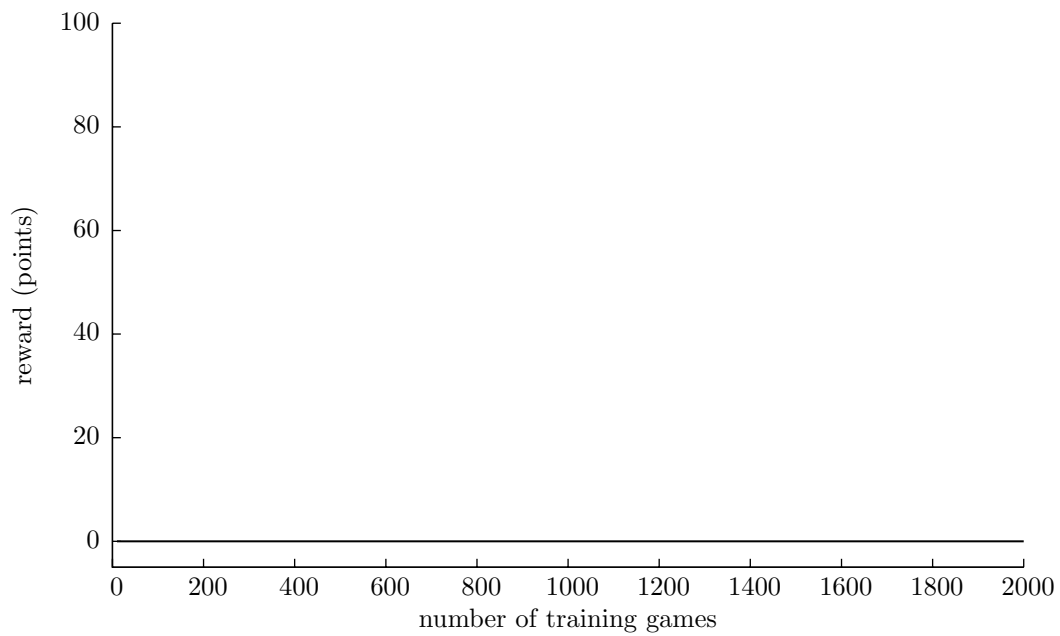


Figure B.40: Training rewards for Wallmaze (white)



# List of Figures

2.1	Feature generation and selection . . . . .	7
2.2	Preference Pair Learning . . . . .	9
3.1	Evaluations of five features using different variable lists and the feature formula $\text{true}(\text{cell}(\mathbb{M}_1, \mathbb{N}, \mathbf{x})) \wedge \text{true}(\text{cell}(\mathbb{M}_2, \mathbb{N}, \mathbf{o}))$ . . .	17
3.2	Overview of the feature generation process . . . . .	18
4.1	Two hierarchies of the same graph . . . . .	43
4.2	Abstraction graph for Tic-Tac-Toe. Features are colored by their matching ratio (darker shades mean higher ratio). All features that have the same level in the abstraction hierarchy are placed on the same horizontal rank. . . . .	48
5.1	Plots of the link function $g(x)$ and its derivative $g'(x)$ . . . . .	53
6.1	Number of new features per expansion depth for the four aborted games (Blobwars, Skirmish, Minichess, Endgame) and the three non-aborted games with the most features (Eight-Puzzle, Wallmaze, Merills) . . . . .	57
6.2	Number of features produced and CPU time spent by each feature transformation, averaged over all 33 games . . . . .	64
6.3	Percentage of average reward received by the learner vs. average of opponents . . . . .	66
6.4	Initial and final positions of the game Crisscross . . . . .	69
6.5	Initial state of the game Wallmaze . . . . .	70
B.1	Training rewards for Eight-Puzzle (player) . . . . .	82
B.2	Training rewards for Asteroids (ship) . . . . .	82
B.3	Training rewards for Blocker (blocker) . . . . .	83
B.4	Training rewards for Blocker (crosser) . . . . .	83
B.5	Training rewards for Bomberman (bomberman) . . . . .	84
B.6	Training rewards for Bomberman (bomberwoman) . . . . .	84
B.7	Training rewards for Breakthrough (black) . . . . .	85
B.8	Training rewards for Breakthrough (white) . . . . .	85
B.9	Training rewards for Checkers (black) . . . . .	86
B.10	Training rewards for Checkers (white) . . . . .	86

B.11 Training rewards for Chinese Checkers (blue) . . . . .	87
B.12 Training rewards for Chinese Checkers (green) . . . . .	87
B.13 Training rewards for Chinese Checkers (red) . . . . .	88
B.14 Training rewards for Circle Solitaire (taker) . . . . .	88
B.15 Training rewards for Connect-Four (red) . . . . .	89
B.16 Training rewards for Connect-Four (white) . . . . .	89
B.17 Training rewards for Crisscross (red) . . . . .	90
B.18 Training rewards for Crisscross (teal) . . . . .	90
B.19 Training rewards for Crossers3 (left) . . . . .	91
B.20 Training rewards for Crossers3 (right) . . . . .	91
B.21 Training rewards for Crossers3 (top) . . . . .	92
B.22 Training rewards for Incredible (robot) . . . . .	92
B.23 Training rewards for Ghostmaze (explorer) . . . . .	93
B.24 Training rewards for Ghostmaze (ghost) . . . . .	93
B.25 Training rewards for Hallway (black) . . . . .	94
B.26 Training rewards for Hallway (white) . . . . .	94
B.27 Training rewards for Merrills (black) . . . . .	95
B.28 Training rewards for Merrills (white) . . . . .	95
B.29 Training rewards for Pacman (pacman) . . . . .	96
B.30 Training rewards for Pacman (inky) . . . . .	96
B.31 Training rewards for Pacman (blink) . . . . .	97
B.32 Training rewards for Peg (jumper) . . . . .	97
B.33 Training rewards for Racetrack Corridor (black) . . . . .	98
B.34 Training rewards for Racetrack Corridor (white) . . . . .	98
B.35 Training rewards for Tic-Tac-Toe (xplayer) . . . . .	99
B.36 Training rewards for Tic-Tac-Toe (oplayer) . . . . .	99
B.37 Training rewards for Tic-Tic-Toe (black) . . . . .	100
B.38 Training rewards for Tic-Tic-Toe (white) . . . . .	100
B.39 Training rewards for Wallmaze (black) . . . . .	101
B.40 Training rewards for Wallmaze (white) . . . . .	101



## List of Tables

2.1	Transformations in Zenith . . . . .	8
6.1	Number of generated, unique and selected features; average computation time needed for the invocation of the resulting evaluation function per state . . . . .	60
6.2	CPU time needed for Feature Generation and Feature Selection . . .	61
6.3	Total number of features produced by each transformation . . . . .	62
6.3	(cont.) Total number of features produced by each transformation . .	63
6.4	Time needed for one training match . . . . .	65
6.5	Average rewards after learning for 400 testmatches . . . . .	67



## List of Algorithms

4.1	AddToAbstractionGraph . . . . .	39
4.2	TransitiveReduction . . . . .	40
4.3	AssignLevels . . . . .	42
4.4	FeatureEligible . . . . .	46
4.5	FeatureSelection . . . . .	47
5.1	RunLearningMatches . . . . .	50
5.2	SelectActions . . . . .	51
5.3	TD Update . . . . .	53



## List of Acronyms

ACF	Always-Changing Fluent (see Definition 3.9)
AI	Artificial Intelligence
DAG	Directed Acyclic Graph
DFS	Depth-First Search
ELF	Evaluation Function Learner (with a twist)
GDL	Game Description Language
GGP	General Game Playing
GLEM	Generalized Linear Evaluation Model
STRIPS	Stanford Research Institute Problem Solver
TD	Temporal Difference



## Bibliography

- Nima Asgharbeygi, David Stracuzzi, and Pat Langley. Relational temporal difference learning. In *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006.
- Bikramjit Banerjee, Gregory Kuhlmann, and Peter Stone. Value function transfer for general game playing. In *ICML workshop on Structural Knowledge Transfer for Machine Learning*, June 2006.
- Jonathan Baxter, Andrew Trigg, and Lex Weaver. Knightcap: A chess program that learns by combining TD( $\lambda$ ) with game-tree search. In *Proceedings of the 15th International Conference on Machine Learning*, pages 28–36, San Francisco, CA, 1998. Morgan Kaufmann.
- Darse Billings, Lourdes Peña, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, January 2002. Special Issue on Games, Computers and Artificial Intelligence.
- Michael Buro. From simple features to sophisticated evaluation functions. In *CG '98: Proceedings of the First International Conference on Computers and Games*, pages 126–145, London, UK, 1999. Springer-Verlag.
- Fredrik A. Dahl. Honte, a go-playing program using neural nets. In Fürnkranz and Kubat (2001), chapter 10, pages 205–223.
- Tom E. Fawcett. *Feature Discovery for Problem Solving Systems*. PhD thesis, University of Massachusetts, Amherst, 1993.
- Tom E. Fawcett. Knowledge-based feature discovery for evaluation functions. *Computational Intelligence*, 12(1):42–64, 1996.
- Tom E. Fawcett and Paul E. Utgoff. Automatic feature generation for problem solving systems. In D. Sleeman and P. Edwards, editors, *Proceedings of the 9th International Conference on Machine Learning*, pages 144–153. Morgan Kaufmann, January 1992.
- Johannes Fürnkranz. Machine learning in games: A survey. (Fürnkranz and Kubat, 2001), pages 11–59.

- Johannes Fürnkranz. Recent advances in machine learning and game playing. *ÖGAI-Journal*, 26(2), 2007. Special Issue on Computer Game Playing.
- Johannes Fürnkranz and Miroslav Kubat, editors. *Machines that Learn to Play Games*, volume 8 of *Advances In Computation: Theory And Practice*. Nova Science Publishers, Huntington, NY, 2001.
- Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.
- Kenji Kira and Larry A. Rendell. A practical approach to feature selection. In Derek H. Sleeman and Peter Edwards, editors, *ML92: Proceedings of the Ninth International Conference on Machine Learning*, pages 249–256, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.
- J. Kittler. Feature selection and extraction. In T. Y. Young and K. S. Fu, editors, *Handbook of pattern recognition and image processing*, pages 60–81. Academic Press, 1986.
- Arsen Kostenko. Calculating end game databases for general game playing. Master thesis, Technische Universität Dresden, October 2007.
- Anton Leouski and Paul E. Utgoff. What a neural network can learn about Othello. Technical Report UM-CS-1996-010, Computer Science Department, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA, March 1996.
- Robert Levinson and Ryan Weber. Chess neighborhoods, function combination, and reinforcement learning. In T. Anthony Marsland and Ian Frank, editors, *Revised Papers from the 2nd International Conference on Computers and Games (CG'00)*, pages 133–150, London, UK, October 26-28 2002. Springer.
- Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford University, March 2008.
- Armand Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.
- Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.
- Arthur L. Samuel. Some studies in machine learning using the game of checkers. II — recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.



- Stephan Schiffel and Michael Thielscher. Automatic construction of a heuristic search function for general game playing. In *Proceedings of the 7th IJCAI International Workshop on Nonmonotonic Reasoning, Action and Change (NRAC07)*, Hyderabad, India, January 7-8 2007a.
- Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In Adele Howe and Robert Holt, editors, *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 1191–1196, Vancouver, British Columbia, Canada, July 22-26 2007b. AAAI Press.
- Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiro Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, 1981.
- Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
- Gerald Tesauro. Connectionist learning of expert preferences by comparison training. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 1 (NIPS-88)*, pages 99–106. Morgan Kaufmann, 1989.
- Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.
- Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1-2):119–165, 1994.
- Paul E. Utgoff and Doina Precup. Constructive function approximation. In Huan Huan Liu and Hiroshi Motoda, editors, *Feature extraction, construction, and selection: A data-mining perspective*, pages 219–235. Kluwer, 1998.



# Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 22. Oktober 2008

---

Martin Günther