Prediction-based Search for Autonomous Game-Playing

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Alexander Dockhorn, M.Sc.

geb. am 17.04.1991 in Halle (Saale)

Gutachterinnen/Gutachter

Prof. Dr. Rudolf Kruse
Prof. Dr. Sanaz Mostaghim
Prof. Dr. Simon Lucas

Magdeburg, den 08.07.2020

Dockhorn, Alexander:
Prediction-Based Search for Autonomous Game-Playing

Dissertation, Otto von Guericke University
Magdeburg, 2019.

Prediction-Based Search for Autonomous Game-Playing

Abstract

Simulation-based search algorithms have been widely applied in the context of autonomous game-playing. Their flexibility allows for the rapid development of agents that are able to achieve satisfying performance in many problem domains. However, these algorithms share two requirements, namely, access to a forward model and full knowledge of the environment's state. In this thesis, simulation-based search algorithms will be adapted to tasks in which either the forward model or the state of the environment is unknown.

To play a game without a forward model, methods for learning the environment's model from recent interactions between the agent and the environment are proposed. These forward model learning techniques allow the agent to predict the outcome of its actions, and therefore, enable a prediction-based search process. An analysis of environment models shows how they can be represented and learned in the form of an end-to-end forward model. Based on this general approach, three methods are proposed which reduce the number of possible models and, thus, the training time required. The proposed forward model learning techniques are evaluated according to their applicability to general game-learning tasks and validated based on a wide variety of games. The results show the applicability of prediction-based search agents for games where the forward model is not accessible.

In case the environment's state cannot be fully observed by the agent and the number of possible states is low, state determinisation methods, which uniformly sample possible states have shown to perform well. However, if the number of states is high, the uniform state sampling approach performs worse than non-determinising search methods due to the search process spending too much time on unlikely states. In this thesis, two methods for predictive state determinisation are proposed. These sample probable states based on the agent's partial observation of the current state and a database of previously played games, which allows the agent to focus its search process on likely states. Proposed algorithms are evaluated in terms of their prediction accuracy and game-playing performance in the context of the collectible card game Hearthstone. Results show that the implemented agent outperforms other state-of-the-art agents in case the replay database is representative for the state distribution.

Zusammenfassung

Simulationsbasierte Suchalgorithmen sind im Rahmen autonom spielender Agenten weit verbreitet. Ihre Flexibilität ermöglicht die schnelle Entwicklung von Agenten, welche in der Lage sind, in vielen Problembereichen eine zufriedenstellende Leistung zu erzielen. Allerdings teilen diese Algorithmen zwei Anforderungen: den Zugang zu einem *forward model*, welches die Zustandsveränderungen der Umgebung bezüglich der Aktionen des Agenten beschreibt, und die vollständige Kenntnis des Umgebungszustands. In dieser Arbeit werden prädikionsbasierte Suchalgorithmen entwickelt, welche ohne die Kenntnis eines *forward models* oder den Zustand der Umgebung verwendet werden können.

Um Spiele ohne die Kenntnis eines *forward models* zu spielen, werden Methoden zum Erlernen des Modells aus den bisherigen Interaktionen zwischen dem Agenten und der Umgebung entwickelt. Diese *forward model learning* Methoden ermöglichen es dem Agenten einen prädiktionsbasierten Suchprozess durchzuführen. Eine Analyse dieser Modelle zeigt, wie sie in Form eines *End-to-End-Forward*-Modells dargestellt und erlernt werden können. Basierend auf diesem allgemeinen Ansatz werden drei weitere Methoden vorgeschlagen, die die Anzahl der möglichen Modelle und damit die benötigte Trainingszeit reduzieren. Im folgenden, werden diese Lernmodelle hinsichtlich ihrer Anwendbarkeit auf *general game-learning* Probleme bewertet und auf der Grundlage einer Vielzahl von Spielen validiert. Die Ergebnisse zeigen die Anwendbarkeit der prädiktionsbasierten Suche für Spiele, bei denen das *forward model* nicht zugänglich ist.

Falls der Agent den Zustand seiner Umgebung nicht vollständig beobachten kann und die Anzahl der möglichen Zustände gering ist, haben sich Zustandsdeterminierungsverfahren bewährt. Mit Hilfe dieser, werden statt eines einzigen Zustands mehrerere mögliche Zustände betrachtet und darauf basierend eine geeignete Aktion gewählt. Ist die Anzahl der Zustände jedoch hoch, schneiden existierende Zustandsdeterminierungsverfahren schlechter ab als nicht-determinierende Suchmethoden, da erstere unwahrscheinlichen Zuständen zu viel Gewicht beimessen. In dieser Arbeit werden zwei Methoden zur prädiktiven Zustandsbestimmung vorgeschlagen. Diese prognostizieren den aktuellen Zustand basierend auf dem bisherigen Spielverlauf und einer Datenbank früherer Spielverläufe. Dies ermöglicht es dem Agenten, seinen Suchprozess auf wahrscheinliche Zustände zu konzentrieren. Die vorgeschlagenen Algorithmen werden im Rahmen des Sammelkartenspiels Hearthstone hinsichtlich ihrer Vorhersagegenauigkeit und Spielleistung bewertet. Die Ergebnisse zeigen, dass der implementierte Agent andere State-of-the-Art-Methoden übertrifft, falls die verwendete Datenbank repräsentativ für die tatsächliche Wahrscheinlichkeitsverteilung auftretender Zustände ist.

Contents

1 Introduction 1

1.1 Background and Research Topic 1

1.2 Problem Definition and Research Questions 9

1.3 Structure of this Thesis 12

2 Basics of Autonomous Game-Playing 15

2.1 The Agent-Environment Interface 15

2.2 A Taxonomy of Autonomous Game-playing 23

3 Algorithms for Autonomous Game-Playing 29

3.1 Heuristics 29

3.2 Knowledge-based Systems 30

3.3 Optimisation / Evolutionary Computation 31

3.4 Reinforcement Learning 33

3.5 Simulation-based Search 35

3.6 Hybrid Models 39

3.7 Other Models 42

3.8 Comparison of Algorithms 43

4 Forward Model Learning 49

4.1 Defining Forward Model Learning 51

4.2 End-to-End Forward Model 53

4.3 Decomposed Forward Models 57

4.4 Local Forward Models 65

4.5 Object-Based Forward Models 69

4.6 Comparison of Proposed Learning Methods 74

4.7 Agent Model 78

4.8 Results of Previous Work 80

4.9 Evaluation Setting 82

4.10 Evaluation of the Prediction Accuracy 84

4.11 Evaluation of the Game-Playing Performance 88

5 Predictive State Determinisation **99**
5.1 Hearthstone: Heroes of Warcraft 101
5.2 Card Sequence Models 106
5.3 Clustering-based Meta-game Analysis 110
5.4 Agent Model 121
5.5 Evaluation of the Prediction Accuracy 122
5.6 Evaluation of the Game-Playing Performance 136

6 Conclusion and Future Work **147**
6.1 Discussion and Research Questions 148
6.2 Future Work 152

Bibliography **155**

List of Figures **179**

List of Tables **181**

A Appendix **185**
A.1 Games of the GVGAI Framework 185
A.2 Grid-Search for Tuning Local Forward Models 193
A.3 Local Forward Model Accuracy per Game 194
A.4 Constant Model Game-Playing Performance 202
A.5 Continuous Learning Game-Playing Performance 212
A.6 Transfer Learning Game-Playing Performance 219

CHAPTER 1

Introduction

1.1 Background and Research Topic

The development of artificial general intelligence is one of the key long term goals in artificial intelligence (AI) research. This includes the development of intelligent algorithms that can handle a variety of problems. While AI methods have produced many successful applications in recent years, the capabilities of AI agents are often limited to very specific problem scenarios. Observing how the human brain is capable of complex reasoning and adapting to new contexts, tasks, and environments, research in AI is still falling behind in developing similarly behaving computer agents.

The research in artificial general intelligence can be divided into the model-oriented and the application-oriented view. In the model-oriented view, the capabilities of the human brain are analysed and partly reverse engineered. Projects such as the Human Brain Project [105] try to reconstruct the entirety of neuronal connections in the human brain. Another approach is the model-based reverse engineering of the brain's capabilities as it is the case with e.g. the COG architecture [70–72] or MicroPsi [9, 10]. These architectures are rooted in a psychological theory of motivation and problem-solving. They have become increasingly complex in recent years, but their applications still seem to be very limited. Evaluating the capabilities of these frameworks is hard due to their promised generality. In contrast, in the application-oriented view, solutions are created by analyzing increasingly complex problems. While each of these solutions is often restricted to a

specific application, the applications have become more advanced throughout the years. The natural advantage of this approach is the comparability of developed frameworks based on the task at hand.

Games, especially digital games, can be useful tools in the development and assessment of artificial general intelligence. Not only do they often have clear and quantifiable goals, but also require complex reasoning processes. Digital games additionally have the advantage to be fully accessible to computers, letting us configure their in- and output without much engineering overhead. Since games are also played by human players, they allow for interesting human-computer-interactions as well as provide us with large data sets of human playing behaviour.

One of the most popular examples of artificial intelligence in games is the development of Chess AI agents. In the early ages of AI research, it was widely believed that playing Chess on a competitive level will be a sufficient feature of general AI. As such, for many years research in computational intelligence in games focused on the development of game AI, which is capable of playing one specific game.

In 1997 IBM developed Deep Blue [31], the first Chess-playing computer agent capable of beating Garry Kasparov, the reigning Chess world champion at that time. Deep Blue combined the knowledge of multiple Chess experts with the computational power of a computer that can search through millions of positions per second. While at this time it was generally accepted that playing Chess is a good way of measuring the capabilities of AI techniques, the proposed solution was more a feat of human engineering than a proof of strong AI. Nevertheless, it remains one of the great milestones in the history of computational intelligence in games.

What began with Chess was continued with the analysis of a wide variety of games. The next major goal became the development of an agent playing the board game Go on a competitive level [25]. While storing the whole game-tree is infeasible with today's memory capacities, the recent success of AlphaGo over the world-champion Lee Seedol proved that computers can still successfully play the game on a human expert level and beyond [158]. The development of AlphaZero [159, 160] further proved that similar game-

playing performance can be reached without access to any human player data. More importantly, AlphaZero's learning generalised well to Chess and Shogi.

Developing a single agent that is capable of playing multiple games became known as general game-playing AI, which seems to be a promising next step in enhancing the capabilities of AI agents [96]. In the study of general games, the agent receives a description of the game in the form of its state and action space as well as the game's ruleset. During the agent development phase, this information is unknown such that the agent itself needs to learn how to play these games effectively. This task requires the agent to combine knowledge representation, learning, reasoning, and decision-making [68], thereby shifting the necessary expertise for playing the game away from the agent's engineer to the agent itself.

The study of general game-learning takes this task one step further by also omitting the ruleset from the description of the game being played. This minor change in the task description has a drastic effect on the agent's learning task. In general game-playing, the agent can use the known ruleset for simulating the outcome of its actions and, therefore, search for suitable actions by analyzing their results. However, general game-learning demands the agent to also figure out the meaning of its actions and the objective of the game while playing it [60].

In summary, the differences of single game-playing AI, general game-playing AI, and general game-learning AI lie in the input given to the agent. Single game AI assumes to have a perfect representation and understanding of the rules at the time of development. In general game-playing AI, the agent's learning task is generalised across multiple games without knowing the meaning of each game's representation. General game-learning AI further removes the knowledge of the rules.

1.1.1 Artificial Intelligence in Games

The current research is dominated by two algorithm classes, namely reinforcement learning and simulation-based search algorithms. Using either in game-based scenarios resulted in many successful applications in board and digital games, including traditional 2-player board games such as Chess, Go, and Shogi [159, 160], partial information games such as Poker [147],

Doppelkopf [49, 157], and Hearthstone [167], and complex strategy video games such as the Civilization [11, 26] and the Total War series [35]. Despite their vastly different concepts, both algorithm classes have resulted in top-performing AI agents. In the following, a short overview will be provided to better understand their differences and their resulting strengths and weaknesses.

Reinforcement Learning Reinforcement learning is a machine learning strategy inspired by behavioural psychology. Its name draws from the concept of reinforcing good actions by providing a reward or teaching to avoid bad actions by punishments. The centre of this framework is a permanent interaction loop between the agent and the environment. With each executed action the environment updates its state and the agent receives some reward. Through continuous interaction, the agent should be enabled to maximise the expected reward by choosing promising actions based on its current context.

Even though the learning strategies for estimating the expected reward vary from algorithm to algorithm, the results have surpassed human expert levels in many applications [115]. However, the training process can be lengthy and requires huge amounts of simulation time and/or computational power. Yet, the training procedure of many reinforcement methods converges to the optimal value function. The necessary time for decision-making is often marginal and only dependent on the number of actions that need to be checked.

Simulation-based Search Simulation-based search algorithms, also known as planning algorithms, do not need to be trained. In contrast to reinforcement learning methods, they estimate the value of each action at run-time. This is done by using the game's ruleset to simulate the outcome of each action. Subsequently, they are ranked by applying a scoring function to their resulting state. The simulation process can be continued similarly until a terminal state has been reached and the scoring function is based on the outcome of each simulated game. Finally, the best-ranked action is applied by the agent.

Similar to reinforcement learning, simulation-based search agents were widely applied in industrial and research contexts. As long as the agent has access to the current state and a simulation method, the so-called forward model, planning algorithms can yield a high performance without a lengthy training process. Still, some of them are proven to converge to the optimal action selection when enough decision-time is available.

Recent Studies on General Game-Playing and -Learning

Simulation-based search algorithms have the common advantage to perform well without training. While reinforcement learning algorithms need time to be trained, which needs to be repeated for every change in the game, simulation-based search algorithms can be applied out-of-the-box. Therefore, they are especially powerful throughout the development phase of a game, during which parameters are bound to change due to balance adjustments.

Because of their flexibility, simulation-based search methods are also known to excel in general game-playing [136]. The General Video Game AI (GVGAI) competition [137] offers a framework for testing and comparing agents on a variety of games. While playing a game in the GVGAI framework, the agent can access its respective forward model. However, the agent is restricted to respond to the current game state in not more than 40ms. Submitted agents are rated based on their average performance in playing each of 5 levels per game 10 times. The low number of trials as well as the short time for decision-making makes it nearly impossible to train an agent during the evaluation.

All the top-performing submissions were based on simulation-based search methods, e.g. Monte Carlo Tree Search [84, 181], Open Loop Search (OLS) [131] as well as the Rolling Horizon Evolutionary Algorithm [62, 63]. Additionally, a recent analysis of the agents' robustness showed that these simulation-based search algorithms tend to cope better with the introduction of noise than planning algorithms such as A^* [135].

The application of these algorithms, however, requires the mandatory availability of two components, namely, a simulation method and a full observation of the current state. To get a better understanding of the impact of these components, scenarios, where either of these is not fulfilled, will be discussed in the following.

Game-Playing in Absence of a Forward Model

As previously discussed, general game-playing without knowing the ruleset, and therefore not having access to a forward model, is also known as general game-learning. In 2017 the GVGAI competition [136] introduced a general game-learning track to compare agents based on how quickly they learn to play simple arcade-like games. Without the provision of a forward model, the agent is allowed to play the first 3 levels of a game for 5 minutes of training time. After this initial training phase, the agents were evaluated based on their performance on two previously unseen levels. Therefore, agents are required to learn how to play a game in a very limited time frame just based on a pixel- or object-based state representation.

Submissions of the game-playing track have shown that simulation-based search algorithms can efficiently be used to play these games. monte carlo tree search (MCTS), rolling horizon evolutionary algorithm and breadth-first search have resulted in a good performance in deterministic games. Due to the inclusion of non-deterministic games, the application of monte carlo tree search variants such as open loop MCTS (OLMCTS) were studied in multiple works (see a summary of agents in [131]). OLMCTS and other tree searching algorithms such as open loop expectimax tree search are able to quickly sample possible action sequences and evaluate their outcome.

The overall performance of agents in the game-playing track is already good in terms of the agents' win-rate and in some cases comparable to the performance of human players. Despite being confronted with a previously unknown game, the agents are often able to win a game or at least find action sequences, which yield a high score. Due to the short training time, the application of reinforcement learning algorithms has not been successful. While these may be able to choose actions much quicker than search-based agents, they also need to be extensively trained before being applied.

Results of the learning track indicate that none of the agents in the past three years of the competition were able to play significantly better than a random agent. Even if it is known that reinforcement learning can learn how to play similarly complex games well (cf. [114, 115]), the short training time makes it nearly impossible to effectively train model parameters based on

the limited data available. Also, the application of simulation-based search methods is restricted to algorithms that try to model the game's features by observation rather than using the forward model.

Despite the good results in the fields of single and general game-playing, submissions to the single-player learning track of the GVGAI competition still lack competitiveness. While algorithms like AlphaZero have shown to be able to play multiple games on an expert level, these deep reinforcement learning-based agents still require enormous amounts of processing power and training time [160]. Results of the single-player learning track have shown that none of the submitted agents is able to outperform a random agent when being limited to just a few minutes of training time. Besides, simulation-based search models that have proven capable of playing unknown games without training time cannot currently be applied without a forward model. Since reinforcement learning methods have already shown that it is possible to learn an approximation of the environment's reward function, it is a promising research direction to do the same regarding the forward model. Learning to replicate the environment's forward model or approximating its result may enable the agent to use simulation-based search algorithms or provide a safe training environment for the training of reinforcement learning-based agents.

Game-Playing in Absence of Complete Information on the Current State

Next to the problem of studying perfect information games, imperfect information games pose additional unique requirements for the application of simulation-based search methods. In a perfect information game, the agent has the information on the initial setup of the environment and all previous events as well as how they changed the environment's state. Imperfect information games, however, hide information from the agent [126], e.g. the distribution of cards in a card game in which the agent does not know its opponent's cards. Since the agent does not know the complete state of the environment it is missing critical information for the application of simulation-based search algorithms.

In such a scenario the agent is forced to determine the current state based on the available information. In this process, called "determinisation", a

hypothetical fully observable state is created in which all unknown variables of the state description are determined by the agent. The generated state can further be used for the application of simulation-based search methods [32].

The algorithm information set UCT [191] uses determinisation to fix several complete information states and runs simulations on each of them. Compatible simulations are combined throughout the simulation phase and a final action is determined by aggregating the results of all simulations. Experiments have shown that information set UCT is unable to outperform determinised UCT in the general domain. Nevertheless, information set UCT excelled in cases where access to hidden information has the greatest impact on determining the results of the game [191]. Another example of determinisation in simulation-based search is ensemble-UCT [157]. Here, the agent determines several complete information games and runs a separate search on each of them. Similar to information set UCT, the agent's action is selected according to the aggregation of each search process. Results of ensemble-UCT have shown to outperform single state-determinisation in case the number of possible states remains low. Experiments on the German card game Doppelkopf showed a significant performance increase in comparison to normal UCT.

A concept that is comparable to determinisation of the current state is the inclusion of chance nodes to handle the determinisation of visited states during the simulation. Here, the current state is assumed to be known and the follow-up state to be determined by a probabilistic process. Tree selection methods such as sparse UCT [20] and UCT+ [28] handle the uncertainty by considering multiple child nodes or inserting chance nodes in case the outcome of a decision is yet unknown. A promising but yet only rarely researched aspect is the inclusion of background knowledge, such as information on the probability distribution of states, in the game state determinisation process. This idea was proposed by Ponsen et al. [141] who used an opponent model for guiding the simulation to nodes with higher probability.

The previously proposed solutions allow the application of simulation-based search to imperfect information games. Both, the inclusion of chance nodes and the ensemble approach, have already shown to benefit the game-playing performance. Nevertheless, the discussed problem domains are

low in complexity in comparison to perfect information games currently being worked on. While the proposed tree selection methods allow the consideration of multiple outcomes, they can also considerably increase the breadth of the search tree in case many options exist. Additionally, reviewing the sampling process shows that a uniform sampling during the state determinisation often results in many simulations being spent on unlikely states. Aggregating the results of multiple unlikely states can result in bad decisions.

As an alternative, background knowledge may be used in accordance with Bayesian inference to derive the current game state based on a database of previously played games and the opponent's previous actions. Studying new methods for incorporating background knowledge, while also considering multiple outcomes to reduce the risk of the decision-maker will be an interesting research direction for improving the applicability of simulation-based search.

1.2 Problem Definition and Research Questions

As previously discussed, simulation-based search algorithms can only be applied in case the environment's forward model and the current state is known to the agent. This work aims to extend the applicability of simulation-based search to scenarios that involve incomplete information on the game state or its rules. The focus of this thesis will be to analyse applications in which either of these two cannot be provided to an autonomous agent and how available information can be used to optimise the agent's behaviour. Throughout this work, predictive models will be used to either learn an approximation of the environment's forward model or to provide a non-uniform state determinisation sampling. These frameworks will be evaluated on a set of real-world problems to test their applicability and study their possible limits. This work will consist of three parts, which will be introduced in the following paragraphs.

Review of Computational Intelligence in Games

In the first part of this work, the state-of-the-art in computational intelligence in games will be reviewed. This review examines the possibilities and

limitations of each method in the context of game-playing AI. Particular attention is paid to a method's restrictions in case the agent cannot access the game's forward model or fully observe its current state. Similarly, it will be reviewed how existing methods and their extensions handle these scenarios. This overview aims to answer the following research question:

Q1 Which methods exist and are applicable in case the agent cannot access a game's forward model or fully observe its current state?

Forward Model Learning

The forward model learning framework will deal with the problem of applying a simulation-based search method without the provision of a forward model. This task was introduced in the context of general game-learning. Here, existing agents were not able to achieve an acceptable game-playing performance in case the forward model cannot be accessed and the training time is limited to a few attempts. In the past, neither simulation-based search agents nor reinforcement learning-based agents were able to significantly outperform a randomly acting baseline agent.

This work will explore the applicability of simulation-based search methods, which are known to perform well out-of-the-box in case access to a forward model can be assured. For this purpose, predictive models will be used to approximate the missing forward model based on the observation of previous interactions between the agent and its environment.

To approach this problem, first, the basic characteristics of forward models will be studied. Furthermore, requirements for forward model learning methods and the application of learned models in the context of a simulation-based search will be discussed. This will allow to qualitatively compare proposed forward model learning frameworks based on their characteristics. The first question that is going to be answered is:

Q2.1 Which characteristics can be used to compare forward model learning processes and their results?

Furthermore, this work will concentrate on the applicability of machine learning methods to learn to approximate an environment's forward model

based on observation. To make this process feasible in the context of general game-learning it will be necessary to discuss how models can be represented and learned efficiently. Therefore, the second question is going to be:

Q2.2 How can a forward model be learned by observation of the agent's interaction with its environment, and how can the model be represented and learned efficiently? To which degree do the proposed models fulfil these criteria?

Finally, the proposed approaches need to be evaluated in the context of general game-learning. How this comparison shall be done will be discussed in the final research question of the forward model learning chapter:

Q2.3 How can the accuracy of forward model learning approaches and their resulting game-playing performance be evaluated? Which of the proposed models performs best?

State Determinisation

In the last part of this thesis, it will be examined how action-selection in the case of a partial state observation can be improved. Therefore, the applicability of predictive models will be studied. Here, it will be assumed that a forward model is available, but the agent only has partial information on the current state of the environment.

Algorithms like information set UCT and ensemble-UCT are basic methods that allow the application of simulation-based search. However, their assumption of a uniform state distribution results in many simulations being spent on analysing unlikely search paths. By improving the state determinisation, simulations could be spent on more likely paths and, therefore, improve the result of the search process.

It was already discussed how the incorporation of chance nodes can benefit the search in non-deterministic scenarios. Thus, it may be beneficial to take the probability of each determinisation into account. For this reason, the first question that is going to be answered is:

Q3.1 How can the probability of each state be determined and how should a state be sampled?

Multiple predictive models will be discussed and compared based on their prediction accuracy. This will be done in the context of the online card game Hearthstone, which offers a large database of game-play data. Using this data-set the following question will be answered:

Q3.2 Can the performance of state determinisation-based search methods be improved by the application of predictive models?

Finally, the best performing model will be used for state determinisation in an ensemble-UCT based agent. The proposed agent will be compared in terms of game-playing performance against a set of state-of-the-art Hearthstone agents to answer the final question:

Q3.3 How does the resulting agent's performance compare to the state-of-the-art?

The result of this work will be a profound analysis of how simulation-based search can benefit from the application of predictive models. Both, the prediction of the current and future states based on a data set of previous interactions, will be analysed in the context of game-playing AI. Resulting methods will increase the applicability of simulation-based search algorithms to a wider range of problems and, e.g. offer game designers powerful tools for the development of AI agents without requiring large amounts of training data.

1.3 Structure of this Thesis

In Chapter 2 the basic terminology of research in autonomous game-playing will be introduced. Thereafter, state-of-the-art algorithms will be reviewed (Chapter 3). The third chapter ends with a comparison of these algorithms with respect to their capabilities and restrictions in case either the environment's forward model or a complete state observation is not provided for the agent. In Chapter 4 and 5, new algorithms for playing games under either of these constraints will be proposed. First, algorithms for autonomous game-playing in absence of a forward model will be adressed in Chapter 4. For this purpose, the properties of environment models are studied to show how an agent can learn to predict the results of its actions and the corresponding changes in its environment. On this basis, the forward model learning

framework and with it four types of forward model representations will be introduced. Chapter 5 begins with an overview of existing algorithms for playing games with a partial state observation. Subsequently, the predictive state determinisation framework will be introduced to specifically address weaknesses. Implementations of both frameworks are evaluated according to their prediction accuracy and game-playing performance at the end of their respective chapters. Chapter 6 presents the answers to the addressed research questions and, thereby, summarises the results of this work. At last, an overview of open research questions and opportunities for future work will be given.

CHAPTER 2

Basics of Autonomous Game-Playing

In this section, basic definitions used in autonomous game-playing will be introduced. The chapter starts with a comprehensive description of the agent-environment interface in Section 2.1 which is commonly used to describe an agent's task. This section is followed by an examination of the individual components of the interface, namely, the environment model, the agent model, and strategies for action selection in the Subsections 2.1.1-2.1.3. Furthermore, several tasks in the field of autonomous game-playing will be distinguished in Section 2.2.

2.1 The Agent-Environment Interface

In this section, the agent-environment interface will be introduced. It offers a universal description of an agent's learning environment in which the agent interacts with its environment to achieve a specified goal [166].

It consists of five components, which are briefly described below:

- **Agent:** The agent is tasked to achieve a predefined goal. It consists of a learning and a decision-making component that aims to select actions such that the agent maximises its chances of achieving said goal.

- **Environment:** Everything that the agent can interact with belongs to the environment. The environment itself can include multiple components which may be observed by the agent. In case multiple agents exist, an agent can perceive another agent as part of its environment.
- **States:** The state of the environment is denoted by $S \in \mathcal{S}$, where $\mathcal{S}$ refers to the state space. A state can consist of multiple values $S = (S^{(1)}, \ldots, S^{(n)})$ which may be observable by the agent. At each point in time t the environment is in a state S_t.
- **Actions:** In this framework the agent can interact with its environment by executing an action $A \in \mathcal{A}$ where $\mathcal{A}$ describes the agent's set of actions. Applicable actions may be restricted by the current state such that the actual action set in state S is denoted by $\mathcal{A}(S) \subseteq \mathcal{A}$. In reaction to the agent's action the environment can change its state and provide the agent with a numerical reward.
- **Rewards:** The reward $r \in \mathbb{R}$ is a performance signal that the agent receives as part of the environment's response.

2.1.1 Environment Model

When performing a task the agent continuously interacts with its environment by applying an action A_t to the current state S_t, observing the future state S_{t+1} of the environment, and potentially receiving a reward R_{t+1}. In general, actions do not need to have a deterministic result. For this purpose, the environment's response (observed state transition and reward) will be described as a stochastic process P over the upcoming response given all previous interactions. This series of interactions can be described as a sequence of states and actions for timepoints 0 to t.

$$P(S_{t+1}, R_{t+1} \mid S_0, A_0, \ldots, S_t, A_t) \tag{2.1}$$

Alternatively, state and reward can be modelled separately:

$$\begin{aligned} &P(S_{t+1} \mid S_0, A_0, S_1, A_1, \ldots, S_t, A_t) \\ &P(R_{t+1} \mid S_0, A_0, S_1, A_1, \ldots, S_t, A_t) \end{aligned} \tag{2.2}$$

In the context of this work, the agent-environment interface will be used to describe games and players. Here, the game will serve as the environment and the player will be represented as the agent. Problematic for the analysis of the environment model is its growing complexity over time since the whole series of previous interactions can be considered. Fortunately, many games exist in which the complexity of their stochastic process is limited due to the environment model's independency of previous interactions. In this case, a game's model satisfies the Markov property [187], meaning:

Definition 2.1 (*Markov Property*)

Let $P(S_{t+1} \mid S_0, \ldots, S_t)$ be the conditional probability distribution of a stochastic process. The process is said to satisfy the first-order Markov property, if and only if the future state S_{t+1} is independent of all states but the present state S_t. Specifically, the conditional probability distribution for the upcoming state is equal to:

$$P(S_{t+1} \mid S_0, S_1, \ldots, S_t) = P(S_{t+1} \mid S_t) \qquad \forall S_{t+1}, S_t \in \mathcal{S} \tag{2.3}$$

Similarly, the process satisties the n-th order Markov property, in case the future state S_{t+1} is independent S_t to S_{t-n+1}

An environment model that satisfies the Markov property can be considered to be a Markov decision process, which is defined by

Definition 2.2 (*Markov Decision Process (MDP)*)

A Markov decision process (MDP) models sequential decision processes, in which at any point in time the decision-maker observes a state S and is asked to provide an action A. By performing the action, the system is put into a new state and can provide a reward to the agent. The MDP is defined by a 4-tuple $(\mathcal{S}, \mathcal{A}, P, R)$, of which $\mathcal{S}$ is the set of observable states, $\mathcal{A}$ the set of available actions, P a transition model, and R a reward function. The transition model and the reward function of an MDP need to satisfy the Markov property.

If the game's transition model and reward function fulfil the Markov property, a game can be considered to be a Markov decision process, in which case the environment's dynamics can be described by

$$\begin{aligned} P(S_{t+1} \mid S_0, A_0, \ldots, S_t, A_t) &= P(S_{t+1} \mid S_t, A_t) \\ P(R_{t+1} \mid S_0, A_0, \ldots, S_t, A_t) &= P(R_{t+1} \mid S_t, A_t) \end{aligned} \tag{2.4}$$

2.1.2 Agent Model

When interacting with the environment the agent's goal is to maximise its reward over time. For this purpose, the agent needs to choose an action depending on the current state of the environment. The agent's action selection will further be referred to as its policy π, which will be defined by

Definition 2.3 (*(Agent's) Policy*)

At each time step t, the agent implements a mapping from the current state S_t to probabilities of selecting each possible action $A \in \mathcal{A}$. This mapping is called the agent's policy π. The term $\pi(A|S)$ describes the probability that the agent executes action A when observing state S.

Executing an action at time step t advances the environment and yields state S_{t+1}. The state transitions of an MDP can be denoted by a state transition graph in which each node represents a state of the environment. Edges mark the transition from one state to another and are annoted with the agent's action and, in case of a non-deterministic environment, by an additional probability of the state transition.

The complexity of a game can be measured by its state-space and its game-tree complexity. The former is measured by the number of states that are reachable from the initial state of the state transition graph. The latter counts the number of paths starting in the initial state and yielding any terminal state. Since this number can be hard to estimate for complex games, the average branching factor (number of edges per node) and the average number of transactions per game (path length from initial to terminal state) can be used to estimate the game-tree complexity.

The performance of an agent largely depends on its policy. In case the whole state transtition graph is known the optimal decision can be determined. However, computing the state transition graph of complex games may be infeasible with current computational ressources. Therefore, strategies for continuously optimizing an agent's policy for a specific MDP are discussed in the following subsection.

2.1.3 Action Selection

One of the main tasks, when studying or interacting with a Markov decision process, is the selection of the best possible action during the current

state to maximise the expected reward over time. Training an agent to do so can either be done in a supervised setting by providing a list of correct actions for each state, or an unsupervised one, in which the agent needs to evaluate an action's value itself. Unsupervised scenarios are often called reinforcement learning tasks since the agent is continuously choosing actions while reevaluating their value given the environment's response. In these, the agent only receives evaluative feedback about the quality of a chosen action, but not about any other available action. Supervised learning tasks provide instructive feedback indicating the best possible action for the current situation. Thus, instructive feedback is independent of the action taken [166].

In the literature reinforcement learning tasks are categorised in associative and non-associative tasks. In a non-associative setting, the agent must find a single optimal action independent of the state of the environment. In comparison, an associative setting is more complex and demands the agent to map every possible input to an output, thus, learning the best possible action for each state. Games that will be studied in the context of this work will be represented as associative reinforcement learning tasks. However, a closer look at how the value of an action can be derived from evaluative feedback in a non-associative environment will allow the introduction of basic action selection strategies. Modified forms of those are commonly used in autonomous game-playing algorithms for associative settings (cf. Chapter 3).

An example of a non-associative scenario commonly studied in the field of reinforcement learning is the *Multi-armed bandit problem* [69]. Consider a casino with n slot machines that reflect various reward distributions. The agent needs to decide which of the slot machines it should play without knowing the distributions at first. Repeatedly playing the same slot machine provides the agent with information on its reward distribution but does not change the state of the slot machine.

Definition 2.4 (*Multi-armed Bandit Problem (adapted from [177])*)

Let a multi-armed bandit have n levers. Given a set of reward distributions $B = \{R_1, \ldots, R_n]$ each being associated with an action A_i, $i = 1, \ldots, n$. Choosing an action A_i is equal to pulling the lever i resulting in a reward value being sampled from its associated reward distribution R_i. The agent's objective is to maximise the accumulative reward over time.

The Multi-armed Bandit Problem is equivalent to a one-state Markov decision process. Since the state of the bandit does not change over time and is independent of all previous actions, each action is directly associated with its reward distribution. Based on the evaluative feedback after each action selection, the agent needs to reevaluate the quality of the executed action. The quality of an action can be estimated by

Definition 2.5 (*Action-Value*)

Consider a non-associative task and let $r_1, \ldots, r_t$ be the observed rewards of the reward distribution R after executing action a for t times. The quality estimate $Q_t(A)$ of an action A at time t can be estimated by

$$Q_t(A) = \frac{1}{t} \sum_{i=1}^{t} r_t = \mathbb{E}(r \mid A) \tag{2.5}$$

With an increasing number of selecting and reevaluating the same action the agent's estimate of this action's value should converge to its true quality $Q^*(A)$ which is equal to the expected value of its associated reward distribution R.

$$\lim_{t\to\infty} Q_t(A) = Q^*(A) = \mathbb{E}(R) \tag{2.6}$$

Maximizing the accumulated reward over time results in a trade-off between exploration and exploitation. If the agent maintains its current estimates of each action's value, then at any time there is at least one action whose estimated value is greatest ($\operatorname{argmax}_{A\in\mathcal{A}} Q(A)$). This process is called a greedy action selection and focuses on exploiting the agent's current knowledge of the actions' values. If the agent selects one of the non-greedy actions, it is called an exploring action which improves the agent's estimate of the action's value. Based on this, the following basic action selection strategies can be derived:

- **Constant ε-greedy:** With $P(\varepsilon)$ pick an action uniformly at random. Otherwise, pick the action with the highest expected reward [166]. The constant ε-greedy action-selection represents a trade-off between exploration and exploitation. A proportion of moves is spent to refine the expected values of actions that are otherwise not chosen by the

greedy agent, thus, ensuring that all $Q_t(a)$ converge to $Q^*(a)$. This can help to identify changes in the environment but comes at the cost of regret each time another option is chosen.

$$\pi(S) = \begin{cases} \text{random action from } \mathcal{A}, & \text{if } \psi < \varepsilon \\ \operatorname{argmax}_{A \in \mathcal{A}} Q(A) & \text{otherwise} \end{cases} \tag{2.7}$$

where ψ is a uniform random number in the range of $[0, 1]$.

- **Decaying ε-greedy:** This method is motivated by the shrinking importance of exploration over the length of the game. At the beginning of a game, the agent has no knowledge of each action's associated reward distribution and needs to repeatedly play each action to explore its value. Over time the agent's estimates will improve and the importance of exploration shrinks. This is represented by reducing the value of ε over time while using a ε-greedy to select the next action leading to more exploitation moves over time.

- **Softmax:** The softmax function chooses an action in proportion to its expected value. In its simplest case, the probability is equal to the ratio of the action's quality and the sum of all actions' quality values.

$$P(A) = \frac{\mathbb{E}(R \mid A)}{\sum_{A' \in \mathcal{A}} \mathbb{E}(R \mid A')} = \frac{Q(A)}{\sum_{A' \in \mathcal{A}} Q(A')} \tag{2.8}$$

A more sophisticated approach uses a Boltzmann distribution to weight differences in estimated quality values:

$$P(A) = \frac{e^{\mathbb{E}(R \mid A)T^{-1}}}{e^{\sum_{A' \in \mathcal{A}} \mathbb{E}(R \mid A)T^{-1}}} = \frac{e^{Q(A)T^{-1}}}{e^{\sum_{A' \in \mathcal{A}} Q(A')T^{-1}}} \tag{2.9}$$

where T is the computational temperature that can be used to balance the trade-off between exploration and exploitation over time.

The presented action selection strategies base their decision on the agent's current estimates of the reward distributions. Non-optimal actions are chosen in favour of exploration. To measure the influence of these non-optimal selections the resulting regret, also called lost opportunity of a selection, can be measured.

Definition 2.6 (*(Total) Regret*)

Given the action-value $Q(A)$ and the optimal value V^* the regret l_t is the opportunity loss of a single decision

$$l_t = \mathbb{E}[V^* - Q(A_t)] \tag{2.10}$$

The total regret L_t measures the accumulated regret of the whole episode.

$$L_t = \mathbb{E}\left[\sum_{t=1}^{T} V^* - Q(A_t)\right] \tag{2.11}$$

Maximising the cumulative reward is equal to minimising the total regret.

Action selection strategies can be compared according to the regret of past decisions and how this number develops over time. If a strategy's average regret per round tends towards 0 with a probability of 1 it is called a zero-regret strategy. These strategies always converge to an optimal action if enough rounds are played. Of the aforementioned action selection strategies only the decaying ε-greedy method is a zero-regret strategy. Applying a decay factor to the softmax functions temperature parameter (cf. Equation (2.9) on the preceding page) can yield a zero-regret strategy.

Being uncertain about an action's value should yield further exploration of this action for improving the agent's estimate of the action's value. Otherwise the agent might regret ignoring this action on the long run. The success of previously discussed action selection strategies depends on an appropriate initialisation. Once the correct action values are known, the greedy action selection produces a total regret of 0. In case an action's value is unknown to the agent or its estimate is inaccurate, the agent needs to choose between exploiting the best known action or exploring the expected reward of other actions.

Upper Confidence Bounds (UCB) is an action selection method that represents this trade-off between exploration and exploitation. At time step

t the agent's action is selected according to the weighted sum of an action's estimated quality and the agent's confidence in this estimate.

$$A_t = \underset{A \in \mathcal{A}}{\text{argmax}} \quad \underbrace{Q_t(A)}_{\text{Exploitation}} + \underbrace{\hat{U}_t(A)}_{\text{Exploration}} \tag{2.12}$$

Here, the agent's confidence in its quality estimate of action A is represented by $\hat{U}_t(A)$. The confidence is high in case an action has been selected many times since more samples of the action's association reward distribution have been observed. The most popular variant of the UCB method is UCB1:

$$A_t = \underset{A \in \mathcal{A}}{\text{argmax}}\, Q_t(A) + C\sqrt{\frac{\ln N(S_t)}{N(S_t, A)}} \tag{2.13}$$

in which $N(S_t)$ describes the number of times the environment has been in state S_t and $N(S_t, A)$ the number of times action A has been selected while the environment was in state S_t. In a multi-armed bandit scenario the state of the system remains unchanged, therefore, $N(S_t)$ is equal to $t - 1$.

The presented action selection methods seem simple, but they form the basis of many algorithms for autonomous game-playing. Before these methods will be reviewed, the different tasks in autonomous game-playing will be described in the following section.

2.2 A Taxonomy of Autonomous Game-playing

Autonomous game-playing addresses the problem of creating an agent that is capable of playing a game with satisfying performance. How the agent's performance is to be measured and at which threshold the agent is fulfiling this goal will be dependent on the task at hand.

A common evaluation is to compare an agent's performance with the performance of human expert players. However, comparisons can also be made between agents to create a ranking. In research on autonomous game-playing agents it is often the goal to achieve optimal performance. This does not need to be a desirable use-case for the gaming industry [187], but it is certainly useful when applying the same algorithms to other contexts.

Research on the topic of computational intelligence in games focuses on developing either an agent that is capable of playing a specific game or describing general techniques for learning to play any game. In the remainder of this work, the tasks single game-playing, general game-playing, and general game-learning will be differentiated. For this purpose these terms will be defined and discussed in the following subsections.

2.2.1 Single Game-Playing

Single game-playing is the most traditional task in computational intelligence in games.

Definition 2.7 (*Single Game-Playing*)

In single game-playing an agent is tasked to play one specific game with satisfying performance. Single game-playing tasks provide a description of the game and its complete ruleset. Depending on the game, the agent tries to optimise its final score or its chances of winning a game.

Much of the research in the field of computational intelligence in games focusses on well-known games such as Chess. Such games were often described as a cognitive task with varying complexity and it was widely believed that solving a game as complex as Chess is an important indicator for the performance of AI agents. Even if proposed solutions of many Chess agents are far from the complexity of human thinking, the research in this field has often strengthened the understanding of the game and the applied methods.

Milestones of single game-playing AI mostly refer to board games, but in recent years many new applications have been seen in digital games. The first program to defeat a world champion in any board game is the backgammon AI BKG 9.8 in 1979 which was written by Hans Berliner [17]. The first agent to win the world champion title in a competition against a human was the draughts agent Chinook. Later the authors had completely solved the game by calculating the best action for all possible game-states [151] and therefore proven that the best achieveable result against Chinook remains a draw. In the meantime, Deep Blue [31] has defeated the chess world champion in 1997. Since then Chess agents have become increasingly stronger. The recent success of AlphaGo [158] and its successor AlphaZero [160] has

shown that a single algorithm can be used to play complex games such as Go, Chess, and Shogi on human expert level and beyond, which marks the transition to general game-playing.

2.2.2 General Game-Playing

As a next step in the direction of general artificial intelligence, it is researched how autonomous game-playing techniques for a single game can be adapted to play any game. Hence, in contrast to training an agent to play a single game, general game-playing focusses on the development of agents, which can learn to play a diverse set of games. To ease the development of such agents the various games are often represented using a unified representation.

Definition 2.8 (*General Game-Playing*)

In general game-playing the agent is tasked to play multiple games with satisfying performance across all games. This task aims at ignoring the actual representation of a game, and therefore, concentrating on a general learning mechanism for playing said games. In general game-playing scenarios the agent can access a description of a game's rules, in the form of a forward model.

Early research on this topic was based on the Stanford General Game-Playing framework [68], in which agents competed in various games that were unknown prior to the agents' submission. Games are descibed using a game description language which consts of logical rules to define state-transitions, actions, and the number of players. Games were often derived The games provided by the framework were often derived from existing board games.

Due to the popularity of video games, similar frameworks have been developed for the study of digital games. One of these frameworks is the general video game AI (GVGAI) framework [132] developed in 2014. Here, games are defined using the Video Game Definition Language [152] which allows the description of 2-dimensional arcade-like video games. At time of writing this work, the framework already provides access to more than 100 games. Since the games were developed especially for this framework, no data from human players is available.

In the related competition, agents are compared on the basis of their performance in several games. Agents can either read the graphical or logical output of the game and provide actions in the form of controller input. Furthermore, the planning track allows agents to access a forward model, which can be used for simulating the outcome of planned action sequences.

Implementing agents to play any game is a complex task. Nevertheless, the agents submitted to the competition have performed well in recent years. With an average victory rate of 50%, the best agents have already shown that it is possible to perform adequately in numerous games. Studying games in which these agents perform bad may uncover open problems and allow for optimisations of applied planning methods.

2.2.3 General Game-Learning

The general game-learning task describes another challenge for the development of autonomous game-playing agents. Similar to general game-playing the agent is tasked to play multiple games. However, the forward model is not provided while doing so. This rather small change in the task drastically limits the agent's options in learning how to play the game and ensures that this needs to be done solely based on its interactions with the game environment.

Definition 2.9 (*General Game Learning*)

General Game Learning tasks demand the agent to play multiple games with satisfying performance without further knowledge of the game or its representation. The agent is tasked to learn how to play the game solely based on its continuous interaction with the game and observing the result of its actions.

Game-playing benchmarks can easily be converted to game-learning benchmarks by restricting the agent's access to background information on the game. At the same time, this allows for a comparison of the performance of game-playing and game-learning agents.

Due to the recent emergence of this field of research, only a small amount of works were published yet. Current research projects often focus on the benchmark problems provided by the Arcade Learning Environment (ALE) and the GVGAI framework. Since 2017 the GVGAI competition includes the

single-player learning track. During the first two years the training time per game was limited to five minutes after which the agent was evaluated by playing two previously unknown levels of the same game. In 2019 agents were allowed to be pre-trained on a set of training games and were evaluated on previously unknown games. While the first variant of this track focused on the agent's ability to perform the same task under varying conditions, the second variant demands the agent to transfer its knowledge gained in the training environments to new environments. As mentioned in the motivation, none of the agents submitted in the past three years of the competition were able to play significantly better than a random agent.

In the following chapter, current algorithms of computational intelligence in games and their relation to the three presented tasks are examined. Since these algorithms, especially simulation-based search methods, do not seem to be suitable for the given problem, it is described why they are currently not applicable and how their applicability can be increased by predictive models.

CHAPTER 3

Algorithms for Autonomous Game-Playing

In the following sections, state-of-the-art algorithms for autonomous game-playing will be reviewed and discussed. The Sections 3.1 - 3.6 each introduce a common algorithm class. Special attention will be given to planning and simulation-based search algorithms in Section 3.5 which form the basis of this dissertation's remaining chapters. This chapter concludes with a discussion about the suitability of presented algorithms for scenarios in which either the forward model is not known or the status cannot be completely observed (Section 3.8).

3.1 Heuristics

A heuristic is an approximate solution to a most often much more difficult problem. In the context of computational intelligence in games, agents can make use of heuristics in all kinds of decision-making problems [111].

In this sense, a heuristic is most often applied as a state-value or state-action-value function, which tries to approximate the game's true value function. Using the heuristic the agent can rate all possible actions and choose the action with the most promising value. Nevertheless, other action-selection methods can be applied as well.

Heuristics have previously been used to guide search processes in general game-playing and -learning. An early work by J. Clune [38] showed the value of heuristic evaluation functions in general game-playing in which an

abstract model was created to represent parts of the original game, namely payoff, control, and termination. Based on these the agent was able to play traditional board games with comparatively low performance. Better results have been achieved by Santos and Bernardino [150] by combining avatar-related information, encouraging spatial exploration, and using obtained knowledge during the agent's game-play to enhance the evaluation of game states. The Yolobot agent [87] for general game-playing and -learning uses a targeting heuristic to guide the search and movement of the agent. Yolobot was ranked as one of the best-performing agents in the General Video Game AI's single-player planning track [136]. However, it did not succeed in achieving a similar performance if there was no forward model, as shown by the results of the GVGAI competition's learning track.

The term hyperheuristics describes a series of heuristics from which the agent can choose at runtime. They have recently been proposed in the context of general game-playing to choose which game-playing algorithm should be applied based on observable game features [109]. The applied hyper-heuristic showed promising results in comparison to the use of a single algorithm.

Heuristics and hyper-heuristics are often baselines for planning, simulation-based search, and reinforcement learning models. In this thesis, these baseline heuristics are considered to be designed by an expert. Methods for learning or optimising heuristics will be discussed in Section 3.3.

3.2 Knowledge-based Systems

A knowledge-based system mimics human decision-making by being provided with a large training set of expert decisions. Supervised-learning algorithms learn to map the input of instances in the training set to recorded decisions made by experts. In the context of game-playing, these training data sets are generated by recording games of expert players. In many cases, it is not possible to generate a training sample for every possible game state. For this purpose, the goal of the supervised-learning algorithm is to build a model that replicates expert moves in known game states and generalises well to unknown game states. Classification al-

gorithms, such as Decision Trees [143], Artificial Neural Networks [73, 88], Support Vector Machines [128], Bayesian Networks [23],, have proven useful in learning to represent and replicate expert moves in games.

This technique is frequently applied to games that offer a large set of gameplay records, such as the traditional board games Chess [36] and Go [117]. Nevertheless, due to the large success of digital games and many efforts of their respective communities, data sets of player data can be found for plenty of digital games, e.g. for Hearthstone[1], Dota 2[2], and Starcraft[3]. Based on such data sets expert knowledge can be extracted to predict multiple game characteristics, e.g. expert action policies [49], likely winners [180], or upcoming deaths [89]. Since such data sets are specific to a single game, learning knowledge-based systems based on human gameplay has not been done in the general game-playing area.

Next to using a knowledge-based system for single game-playing, analysing learned models may also uncover new patterns that were unknown to the experts that trained the agents model. Hence, generating an interpretable model can be useful for human-computer interaction as well. Learning association rules to describe unknown games has shown to be a useful technique for communicating a game's ruleset and termination conditions to human observers [54]. Similar systems can be applied in the context of general game-learning for describing either the game or the behaviour of an agent based on replay data.

3.3 Optimisation / Evolutionary Computation

Optimisation schemes can be applied to a wide area of tasks. Most popular in the context of game AI is the optimisation of action policies or scoring functions, which both can be represented in the form of heuristics.

The simplest class of optimisation algorithms is local search. Here, an objective function is optimised by constantly adapting and reevaluating a single candidate solution. Optimised methods such as hill climbing or simulated annealing can speed up the optimisation process and reduce the chance of getting trapped in a local optimum [92].

[1] http://www.hearthscry.com/CollectOBot
[2] https://archive.ics.uci.edu/ml/datasets/Dota2+Games+Results
[3] http://archive.ics.uci.edu/ml/datasets/skillcraft1+master+table+dataset

Particle swarm optimisation generalises this concept by optimising multiple candidate solutions (called particles) at the same time. Each of these particles is adapted according to its position and velocity, but can also take into account the best solution found by other particles. Due to this, the particles are expected to quickly move to the best solutions and search for optimised parameters in promising areas of the solution space. A comprehensive survey of recent trends in particle swarm optimisation was written by Zhang et al. [190].

Recent examples of particle swarm optimisation in agent development include the optimisation of neuro-controllers for playing Snake [176] and the collective behaviour design of non-player characters in first-person shooter games [44]. A survey by Jithesh et al. further summarises the role of particle swarm optimisation in games on a broader scale [86].

Another popular type of optimisation algorithms is the class of evolutionary algorithms. Similar to particle swarm optimisation, evolutionary algorithms optimise a set of candidate solutions. This is done by adapting single individuals through mutation operators and recombining multiple individuals using crossover operators. A comprehensive summary of evolutionary algorithms can be found in [92].

Next to optimising heuristics, evolutionary algorithms are also frequently used in the form of genetic programming. In the context of game-playing agents, this can involve the generation of action plans [66] or action sequences [106].

Evolutionary algorithms and genetic programming have played an important part in the development of game AI. Studies have shown that these techniques can be used to create micro [5, 97, 170] and macro strategies [12], game state evaluation functions [121], as well as complex bot behaviour [1, 116, 192] for single game-playing. The latter has been achieved using genetic programming for generating complex action plans. Similar experiments have been conducted on the evolution of simple controllers for playing Atari games [192]. Finally, the evolution of hyper-heuristics for general game-playing was proposed in a work by Azaria et al. [8]. Next to the development of AI agents, genetic programming has also seen applications in level generation [118], which in turn can be used to generate a diverse set of training levels.

3.4 Reinforcement Learning

During reinforcement learning the agent is taught to pick actions for maximizing a cumulative reward. Given a policy π the state-value function $v_\pi(s)$ describes the objective value of a state s and is defined by its expected return:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \tag{3.1}$$

The action-value function $q(s,a)_\pi$ for policy π, is the expected return starting from s taking the action a and further following the policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t|S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{3.2}$$

Both functions can be estimated based on the observed reward signal during the agent's interaction with its environment. While Temporal Difference Learning [165] updates the agent's value function after every observed reward, the Monte Carlo method [148] updates according to the result of each finished episode. A variant of Temporal Difference Learning, namely Q-learning [182], operates on the agent's state-action-value function based on recent interactions. These techniques are called model-free since they do not try to build a model of their environment. In contrast, Dynamic Programming [83] uses the game's model to update the expected return based on the weighted average of all successor states' values.

Due to its generality, reinforcement learning is a promising framework in terms of general game-playing. However, it can be quite limited regarding the number of feasible states and actions. During learning it is recommended to visit each state several times to test out the various actions [111], therefore, being dependent on the size of the state and action spaces. In case the number of states or actions is high, this process can take many iterations to get an appropriate estimate of the value function. This is especially troublesome in case of a continuous state or action space. Here, first an internal state representation needs to be created by discretizing these

spaces and perform the training mechanism on the internal representation. Such internal state representations often include hand-crafted features to enhance the performance of the algorithm [187]. Changing the state representation may influence the time needed for training as well as the training's outcome [101].

Reinforcement learning algorithms allow their application in an offline learning setting, i.e. once a model is trained, it can be deployed to other machines. Even if the learning process can take a very long time to converge, the time taken for applying the learned model is often negligible. Therefore, the complex models can be trained using large-scale machines, while the final model can be used on less powerful systems.

A remaining limitation can be the memory requirement of these methods. Since the estimated value of each state or state-action pair needs to be stored, the methods seem inapplicable for complex state spaces. This seems to make it impossible to apply reinforcement learning to games such as Go, which state space is roughly 10^{170}. Storing all state-values seems to be impossible[4]. In terms of general game-playing, this could be especially problematic in complex state representations such as the video output of the game. Nevertheless, both have been shown to be approachable with reinforcement learning, using a technique called deep reinforcement learning.

In contrast to simple Q-learning, which is storing the q-value of each state-action pair, deep reinforcement learning is using a neural network to approximate the q-value based on the input, thus drastically decreasing the storage used for the model [187]. In past years deep reinforcement learning (also known as Deep Q-Learning) received special attention for its ability in surpassing human experts in Atari 2600 video games provided by the ALE environment [15]. The team of DeepMind Technologies showed that their agent is capable of learning to play Atari 2600 games by repeated play only using the pixel output of an Atari emulator [114, 115]. Here, a neural network is trained to estimate the Q-values of each action given the preprocessed image. Similar techniques have been applied to games with more complex state representations. In the game Doom (a 3D first-person shooter) deep reinforcement learning was used to execute various tasks using

[4]compared to the estimated number of atoms in the universe, which is about 10^{78} to 10^{82} according to `https://www.universetoday.com/36302/atoms-in-the-universe/`

a visual state observation [57]. Deep reinforcement learning has further been applied to games of the GVGAI framework [171]. Here, the results differed a lot in between tested games. In general, planning agents seemed to outperform deep reinforcement learning agents as long as they had access to the games' forward models.

3.5 Simulation-based Search

Autonomous game-playing can be achieved through a variety of algorithm classes, of which the class of simulation-based search algorithms has been one of the most successful in recent applications. Since simulation-based search algorithms will form the basis of later chapters, they will be described in more detail.

Simulation-based search algorithms use the environment's forward model to simulate the outcome of hypothetical action sequences. Simulating a single action sequence is called rollout. Aggregating the result of multiple rollouts can be used to estimate the value of an action at runtime. Therefore, these algorithms require knowledge of the current state and the game's model as an input and return the action with the highest expected value regarding the current state.

The value of an action can either be the expected chance of winning the game or the number of points scored after applying the action to the current state. Starting at the current state the search method performs multiple rollouts. Once a terminal game state has been found, its result is backpropagated along the simulation path, such that the expected value of each applied action at its current state is updated according to the result. Thus, the expected value of each action can be determined according to the result of performed simulations.

3.5.1 Exhaustive Search Methods

Search methods differ in the way they analyse the game tree and aggregate the result of performed simulations. The most basic algorithm is an exhaustive search [166] in which each possible sequence of actions leading to terminal states is simulated. Therefore, analysing the whole game-tree. A popular

variant for two-player zero-sum games [138] is the minimax algorithm [107] in which both players try to maximise their chances of winning the game. Pruning methods, such as alpha-beta pruning [166], can be applied to reduce the number of game tree nodes to be explored. This algorithm has been proven to be optimal for trees and randomly assigned leaf values in terms of expected run time [127].

Minimax search in conjunction with alpha-beta pruning was successfully applied to games with a large number of states such as chess [42]. However, the performance declines if the number of states is further increased. A popular use case is the board game Go. Here, an exhaustive search is infeasible, due to the size of Go's state space ($\approx 10^{170}$ states [172]). As a result classical tree search methods were outperformed by heuristic tree searches [117].

3.5.2 Flat Monte Carlo

In contrast to exhaustive search, heuristic search methods do not explore the whole game tree, but only sample parts of the game-tree to determine an approximation of each action's value. This concept is reflected in the flat monte carlo algorithm. For each action, it determines the conditional probability of winning the game by simulating a set of action sequences starting with said action [29]. The average score or the agent's win-rate per action serves as an approximation of an action's value. Similar to the analysis of probability distributions, the confidence in the estimated value can be improved by continuing the sampling process.

Using a large number of rollouts the agent may be able to estimate each action's value with sufficient accuracy even if parts of the game-tree remain unexplored. However, a uniform number of simulated episodes per action may not be the most efficient way to identify the best action. Let us assume an action whose approximated value is inferior to other actions' values. Simulating further episodes starting with the inferior action may waste computational resources, which may have better been spent to further differentiate the other actions [90].

Similar to the discussed action strategies a minimisation of regret can be achieved by analysing the trade-off between exploration and exploitation. Instead of simulating the same number of episodes for every action, an action selection strategy can be used to determine the next action to be

simulated. The application of the zero-regret strategy UCB results in a faster convergence than other techniques. The resulting algorithm, called Flat-UCB, is much better in computational efficiency, but can still be improved by reusing results of its simulations more effectively.

3.5.3 Monte Carlo Tree Search (MCTS)

An apparent problem of Flat Monte Carlo and Flat UCB is that executed rollouts only increase our confidence in the first actions' values. At the same time, the result could have been used to estimate the value of each state visited during the rollout. However, storing all these values can exhaust our memory capacity in case the game tree is very large.

The monte carlo tree search algorithm handles the trade-off between storing information on visited nodes and minimizing the required storage by implementing a two-phase search process. For this purpose, a search tree is built in which each node represents a state of the environment. Tree nodes store the number of rollouts which included this node and the approximated value of its represented state. To built the tree, a tree policy is used to choose a node for expansion. The tree policy balances exploration and exploitation by taking the number of simulations started from each node and their resulting approximated value into account. Consecutively, the default policy is used to perform one or multiple rollouts and update the tree nodes' estimated value. During the rollout, actions are often randomly picked to maximise the number of simulations. Information on nodes visited during the rollout is not stored, but the result of each rollout is backpropagated along paths visited during the tree policies selection process. This way, MCTS increases the confidence of actions represented in the first layers of the tree while keeping the memory consumption low.

The MCTS algorithm can be structured in 4 steps which are visualised in Figure 3.1 and explained in the following:

1. **Tree Selection:** The tree policy is used to find an expansion node. Similar to Flat-UCB, the MCTS algorithm tries to minimise the potential regret of our action selection while maximizing our confidence in the value of each tree node. The combination of MCTS using upper confidence bounds as a tree selection policy is also called the Upper

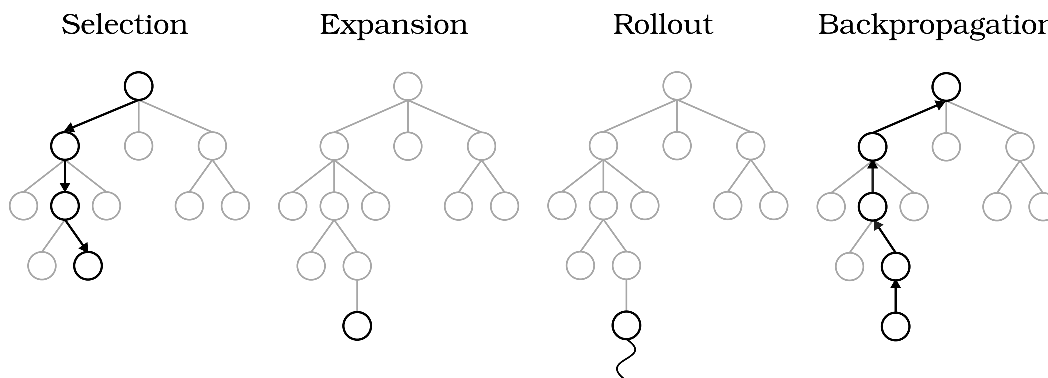


Figure 3.1: Steps of the monte carlo tree search algorithm. Adapted from [29]

Confidence Bounds applied to Trees (UCT) algorithm [90]. It gained a lot of attention due to its great performance and its theoretical convergence to the minimax algorithm [163].

2. **Expansion:** After selecting a node using the tree policy, the node is expanded by adding a new child node to the agent's search tree. To initialise the value estimate of the newly added tree node the search continuous by performing a given number of simulations in step 3.

3. **Monte Carlo Simulation/Rollout:** During the simulation the default policy is continuously applied for quickly choosing the next action until a terminal state has been reached. In its simplest case, a uniform random selection of moves can be used to quickly choose actions. Nodes visited during the simulation are neither stored in the search tree nor evaluated.

4. **Backpropagation:** After a terminal state has been reached, its result is used to update the value estimate $Q(s, a)$ and number of performed simulations $N(s)$ for every node visited during steps 1 and 2.

Similar to flat monte carlo, monte carlo tree search is an anytime algorithm, meaning it can be stopped after every simulation and provide the agent with the action that is currently believed to be the best choice. MCTS can be applied without a scoring function since the return calculation can be based on the result of the simulated episode. In its most basic form, it is

based on the ratio of wins and losses that occurred during the simulations. For games with deep game-trees, MCTS can be used in conjunction with a scoring function to stop simulations early and rate intermediate game-states.

Another method that has attracted a lot of attention is the development of improved default policies. This has shown to reach similar or better playing performance, while considerably reducing the number of necessary simulations. Nevertheless, it introduces another trade-off between the quality and the speed of performed simulations [124]. A comprehensive review of MCTS and its many extensions can be found in [29].

In deterministic applications, nodes of the search tree represent states of the environment. The edges in between nodes describe actions that lead to a state transition from the source to the target node. This is especially useful while traversing the tree during the tree policy stage since the evaluation of an action sequence may already be stored in the tree and does not need additional calls of the forward model. Nevertheless, non-deterministic processes can yield different results for the same action sequences. As a result, storing the state in the node cannot assure that the sampled state is representative of all possible results of the same action sequence.

The Open Loop Search (OLS) [131] algorithm solves this by not storing the resulting state of an action sequence in its node, but only the statistics of all its simulation results. Therefore, the open loop case stores statistics on action sequences and not on state-action pairs (known as a closed loop).

3.6 Hybrid Models

While the previous sections presented basic algorithmic frameworks for game-playing, the following paragraphs shortly discuss hybrid approaches commonly found in the literature.

3.6.1 Evolutionary Algorithms and Search

Evolutionary algorithms have frequently been used in conjunction with planning and search algorithms. Here, three common approaches exist, namely, evolving a scoring function, online-optimisation of search parameters, and

generating action sequences using an evolutionary algorithm. The latter has resulted in the rolling horizon evolutionary algorithm (RHEA), which is shortly discussed below.

Rolling horizon methods are search processes with limited search depth. They produce a set of action sequences of the same length (called horizon) and apply a forward model to calculate and score their resulting state using a heuristic function. At the end of the search process, the first action of the most promising action sequence is applied. Research on rolling horizon methods is motivated by the increasing focus on real-time aspects of games and other applications. These applications require the agent to converge fast without much computational overhead.

This general concept of the rolling horizon was applied in multiple research works. Their common baseline is the rolling horizon random search algorithm which randomly generates action sequences for evaluation. This scheme was improved by the application of evolutionary strategies for generating new action sequences. In general game-playing, the RHEA is known to produce good results on the GVGAI benchmark [62, 63].

The evolutionary generation of action sequences has also been used in conjunction with monte carlo tree search [99, 130, 139]. Specifically, applying uniformly random rollouts has been considered to be uninformative, since many rollouts are wasted on unlikely search paths. Here, an evolutionary algorithm was used to produce rollouts of higher quality, resulting in increased efficiency of the search process. Other boosts in efficiency have been achieved by online-optimisation of the search parameters [161, 162], and the evolution of game-specific heuristics in single [2] as well as general game-playing [16].

3.6.2 Reinforcement Learning and Search

The combination of reinforcement learning and search algorithms has recently gained a lot of attention due to the success of AlphaGo [158]. The algorithm used in AlphaGo combines deep reinforcement learning with a guided search to avoid weaknesses of both method classes. Here, two neural networks were trained, a first network to predict likely moves of expert players to guide the search process and a second network to rate a board position to allow for early stopping. The system was further adjusted

through repeated self-play. The final version of AlphaGo was able to defeat world champion Lee Sedol. It was later shown, that the self-play training mechanism is in itself able to produce similar results without making use of any expert player data [159, 160]. The resulting game-playing agent called AlphaZero was able to beat state-of-the-art agents in chess, shogi, and go.

Reinforcement learning has further been used for the optimisation of rollout policies of MCTS [84]. Given enough training time, this process may be able to uncover useful macro actions and therefore speed-up the search considerably. However, the agent was not able to outperform a random agent, which is likely due to the tight training time limits of the GVGAI competition's learning track, on which this agent has been tested on.

In another concept, reinforcement learning has been used to identify a set of valuable options per state. These options can further be used to guide the succeeding search process. Similar to policy learning, the agent has shown to be able to identify useful actions depending on its current state [140]. After training the agent for thousands of matches it was able to beat other agents in the context of a fighting game. However, methods for faster adaptation seem to be necessary to create more competitive fighting AIs [140].

3.6.3 Ensembles

The basic principle of the ensemble methods is summarised in Condorcet's jury sentence. It states that the combination of independent predictors achieves a higher prediction accuracy than their individual predictions. Ensemble machine learning has seen many applications [189] and is known to improve the precision over the use of single action-selection algorithms.

This idea has been applied in single-game AI through the combination of multiple knowledge-based systems, e.g. an ensemble of neural networks [168] or Bayesian regressors [175]. Fern and Lewis have shown that monte carlo tree search also benefits from an ensemble approach [59], due to better parallelisation on multi-core machines.

Using ensembles in general games were studied by Bontrager et al. while trying to match games with specialised algorithms [22]. Here, the idea is to create a meta-algorithm that identifies a game's type (e.g. puzzle, racing, shooter) and chooses an appropriate algorithm accordingly. The no-free-

lunch-theorem comes to mind here, which states that for the entirety of all problems there cannot be a universal method that solves them best [184]. Therefore, it would always be possible to construct a game in which a specific meta-algorithm fails to find an appropriate match. This problem has recently been studied in the context of the general video game AI framework [7]. The authors suggest that the generality criterion of the no-free-lunch-theorem may not be fulfilled in the context of games since many games that can be theoretically constructed would be of no interest to a human player. Finally, the implementation of such an ensemble system has been shown to result in a better game-playing performance than the application of single algorithms [4, 7].

3.7 Other Models

Aforementioned algorithms have been widely applied in research on computational intelligence in games. However, other methods exist which are part of the game designers toolbox and, for the sake of completeness, will be described in the following paragraphs. These techniques are frequently applied in the industry to describe the behaviour of non-player characters. While they can be efficient in simulating intelligent behaviour they cannot be considered AI methods since they lack any kind of learning and freedom during the decision-making process of the agent.

3.7.1 Finite State Machines

Finite state machines consist of a set of agent states and a set of actions. Actions are selected according to the state of the agent and its environment. As a results the agent's state can be changed according to external inputs.

Usually a graph-like representation is used to design and visualise finite state machines. This makes it a simple yet powerful tool in the hands of a game designer. Nevertheless, state machines can yield predictable and static behaviour which will remain unchanged after their design is finished. To overcome these drawbacks, pre-conditions and action descriptions can be extended by, e.g. fuzzy-logic [142] and probability theory [187].

3.7.2 Behaviour Trees

Behaviour trees [33, 34, 85], which are frequently used in industry, cannot perfectly be ordered into the previously named categories. While their application can yield complex and seemingly intelligent behaviour, behaviour trees do usually not change over time, resulting in predictable behaviour. However, they remain a typical choice in the field of game design, due to their flexibility and interpretativeness.

While often being designed by hand, behaviour trees have also been optimised through machine learning methods. The application of learning behaviour trees has resulted in dynamic behaviour and more reactiveness to the human player [169]. Evolutionary approaches have further been used to automatically optimise pre-constructed behaviour trees for certain navigation tasks [123]. Finally, behaviour trees have been created using genetic programming to play previously unknown levels of jump and run games [39]. Resulting trees were rather low in complexity, but still able to learn to navigate through levels of varying difficulty in a reasonable time.

In the context of commercial video games more complex agent models have been developed using planning systems [122]. Here, a planner describes a high level instance that sets a goal and creates a plan how to reach that goal given the observation of its current environment. A frequent re-evaluation of the current plan allows the agent to keep reactive to its environment. The created plan is later performed using a low-level component such as finite-state machines or behavior trees. The combination of planning and execution has been successfully used within several game-related research frameworks, e.g. microRTS [120] and Fighting Game AI [119].

3.8 Comparison of Algorithms

The presented algorithms are compared with regard to their training methods and their suitability in scenarios which either do not provide a forward model or only a partial state observation.

The first comparison focusses on the training and evaluation process of presented algorithms. Reinforcement learning algorithms represent an eager learning process which means that the training process results in a final estimate of each state's value. Similarly, knowledge-based systems build

a model which replicates human actions without evaluating the action's value. Therefore, in eager learning processes, further evaluations become unnecessary once the model has been built. Nevertheless, the training time can be long for complex games.

In contrast, simulation-based search algorithms represent a lazy evaluation approach which does not require any training. Instead, the agent evaluates each action's value by analysing their possible results at run-time. The time needed for this evaluation depends on the size of the game-tree and the desired accuracy of the estimate. This can become a limiting factor in the agent's performance if the time for action selection is limited.

Hybrid models, in which reinforcement learning and simulation-based search algorithms are combined, represent a mix of eager learning and lazy evaluation. This becomes especially useful in case the eager learning process would require too much training time and the lazy evaluation would be inaccurate due to the number of explorable game-states. This can be done by using reinforcement learning to continuously adapt the default policy for improving the efficiency of the search process.

The second comparison is based on the algorithm's knowledge of the environment. Given the agent's action and the current state, the environment will produce two outputs: the next state and the reward. Both can be encoded in a separate model, namely the forward model producing the next state and the reward model which in turn provides the agent with a reward. However, since the accumulated reward (also known as return) is much more useful for the agent's action-selection process, the following comparison will use the agent's knowledge of the return. Therefore, agents will be reviewed according to the two dimensions, being aware of the game's forward model and knowing the return value of each state.

Reinforcement learning and simulation-based search methods once again represent the extreme ends of this comparison. The eager learning process of reinforcement learning methods results in a return model. While temporal difference learning and the Monte Carlo method do this without needing knowledge about the forward model, the computations done in dynamic programming need the forward model for its iterative update routine. Deep reinforcement learning replaces the need for storing the expected return for

every state by learning a network that approximates the return function. The reduction in complexity can arguably be achieved through an understanding of the game's state-space.

In contrast, simulation-based search methods do not need to store the expected return of each state, since they approximate at run-time using the forward model. The rolling horizon evolutionary algorithm seems to be a slight exception to this, since it is also making use of a heuristic function, which approximates the value of a state. This heuristic function is being used to rate a rollout's outcome and, in the best case, approximates the expected return.

One approach that will be proposed in the context of this thesis is called forward model learning (see Chapter 4). Its goal is to allow the application of simulation-based search in scenarios in which the forward model is inaccessible. This should be achieved by learning a forward model by observation, which can then be used as a replacement for the missing original model. Since in the original model, the reward and the following state are often connected, the applied techniques may also be able to capture knowledge about the reward distribution.

This process can be compared to recent experiments on world modelling [74] and imagination-based deep reinforcement learning [183]. In these, the agent uses a deep or recurrent neural network structure which handles the selection of the agent's actions according to predicted future states. Agents using these model-based deep reinforcement learning approaches have shown to be capable of playing games based on their visual state representation [80]. Similarly, they achieved improved performance in several visual control tasks in comparison to model-free reinforcement learning agents [75]. However, the sheer number of parameters to be tuned and the eager learning of the state's value function results in the agent requiring lots of training data, e.g. $1 \cdot 10^8$ training steps for learning to play the game Sokoban [183]. The amount of required iterations makes this process infeasible in case the model's training time is limited. In contrast, a prediction-based search can be implemented which determines an action's value according to simulations of the trained forward model. Furthermore, reductions of the feasible model space could be achieved by assuming inde-

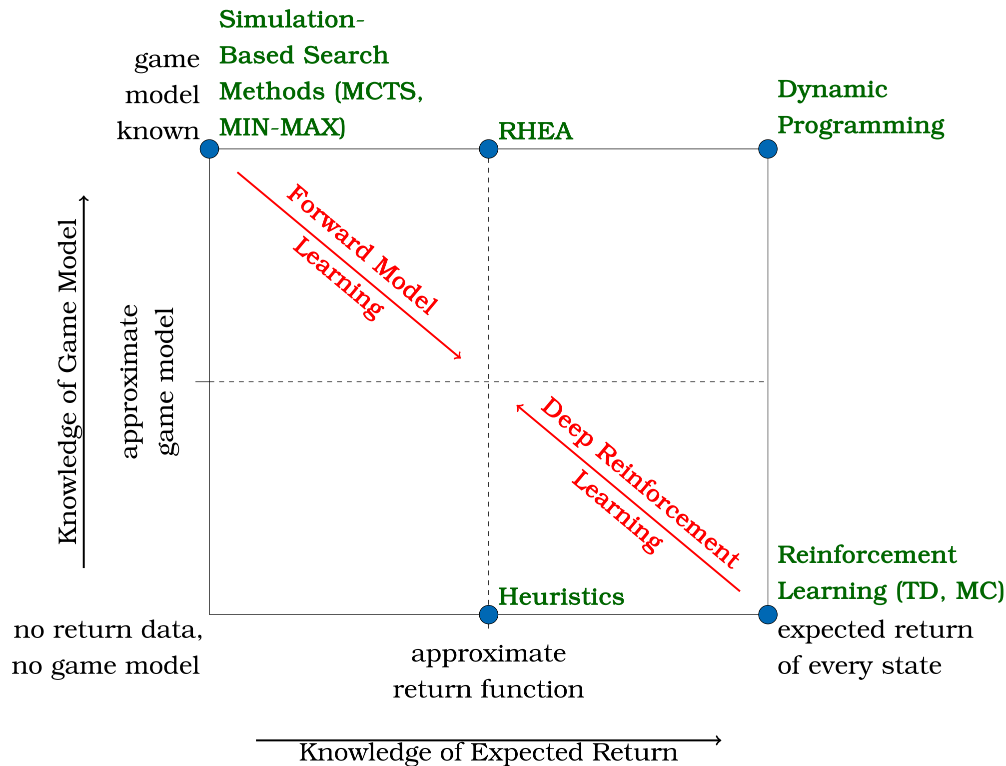


Figure 3.2: Comparison of general game-playing and -learning techniques based on knowledge of the return function and the game's model.

pendencies among observed sensor values. Both, a prediction-based search agent and methods for the efficient representation of forward models, will be proposed in Chapter 4.

Figure 3.2 presents a summary of the discussed methods, based on the two dimensions: knowledge of expected return and knowledge of the game model.

The second problem scenario that this thesis will focus on is the agent's action selection based on a partial state observation. Heuristics and knowledge-based systems are hardly affected by this limitation, since the agent's developers need to handle this restriction. However, more efficient heuristics may be implemented in case of a complete state observation.

Similarly, reinforcement learning agents can learn to estimate the expected return based on a partial state observation [79]. However, a state's true value can be dependent on hidden components of the state observation. As a results, the esimate of the partial information state's value needs to take possible values of hidden state components into account.

Simulation-based search algorithms use the information on the current state and the forward model to simulate the outcome of an action. In case the future state is dependent on information that remains hidden due to the partial observation, no simulation can be performed. State determinisation algorithms sample a hypothetical complete information state which can further be used during the search process. These have shown to perform better than non-determinising search processes in case the number of possible states remains low [50, 191]. How this approach can be extended to cope with larger state spaces will be discussed in (Chapter 5).

Chapter 4

Forward Model Learning

The absence of a forward model drastically changes the way in which computational intelligence agents can approach problems. This becomes evident when comparing agents that have access to such a model to agents that are not able to simulate future steps. Such a comparison can be made for agents that competed in the GVGAI competition's single-player planning track and agent's that entered the single-player learning track (cf. section 1.1.1). The competition results of the years 2017-2019 show drastic performance differences between agents of the two competition tracks. Agents of the learning track often failed to learn how the game can be played and were incapable of winning most games. Furthermore, their performance is often comparable to or worse than the performance of an agent acting at random. This is reflected in the random agent becoming the second-best performing agent in the 2017 instance of the GVGAI competition's learning track [133].

Until recently, the absence of a forward model required the agent to use a model-free learning approach (e.g. reinforcement learning in Section 3.4). Using this type of methods the agent learns to estimate the value of each action in a specific situation. When being confronted with an unknown situation, the agent either does not know the value of its actions (TDL or MCM) or the agent's approximated value function may be wrong due to over- or underfitting (Deep Reinforcement Learning). Experiments have shown that these methods can produce human competitive results when given enough training time. However, such an eager learning approach has shown to not

do well in case the training time is restricted. This is reflected in the minor performance gains over a random agent in the reinforcement learning-based agents of the 2018 and 2019 GVGAI competitions' learning-track [133].

In this work, an alternative method will be proposed which is based on the idea of learning to predict changes in the environment instead of approximating their expected return function. Under the assumption that the outcome of observed interactions can be a predictor for the result of future interactions, learning a forward model may enable an agent to make predictions of the environment's future state. The outcome of this may allow the agent to apply simulation-based search methods and result in faster performance gains than reinforcement learning methods can offer. The resulting lazy evaluation of action sequences may allow the agent to act in unknown environments without collecting an excessive amount of training examples to approximate the environment's return function.

The following section (Section 4.1) presents a general analysis of forward model characteristics in the context of games. Specific implementations of forward model learning will be proposed in subsequent sections. The end-to-end forward model (Section 4.2) presents a general framework for forward model learning. The following, more specific models will be introduced in Sections 4.3-4.5. These aim to reduce the learning effort by restricting the model's hypothesis space based on assumptions on the underlying state representation. A qualitative comparison of proposed models will be presented in Section 4.6 after which the forward model learning-based agent model will be introduced in Section 4.7. Section 4.8 will present a summary of forward model learning experiments scattered throughout previous work. In this work, specifically, the applicability of forward model learning techniques in the context of general game-learning will be evaluated. The evaluation setting is introduced in Section 4.9 which consists of a diverse selection of games of the GVGAI framework. Proposed methods will be evaluated according to their prediction accuracy (Section 4.10) and their game-playing performance (Section 4.11).

4.1 Defining Forward Model Learning

A forward model *fm* (also called environment model) maps the history of previous interactions between the agent and the environment to the next state of the environment. In case the environment is a first-order Markov process, the upcoming state only depends on the latest interaction. Such Markovian forward models form an interesting subset and can be frequently observed in the context of games. For the sake of simplicity, only Markovian forward models will be considered in the following sections. How the proposed methods can be extended to Markov processes of higher-order will be discussed in Subsection 4.2.2.

Proposed forward model learning methods will be based on the following definition of a forward model of 1st order Markov processes:

Definition 4.1

Given an environments state S_t at time t and the agent's action A_t a forward model fm describes the mapping:

$$fm : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathcal{S} \qquad (S_t, A_t) \longmapsto S_{t+1} \tag{4.1}$$

The task of learning or approximating a forward model can be considered a supervised-learning task. Its goal is to learn a model which can predict the upcoming state $S_{t+1} \in \mathcal{S}$ given a history of previous interactions. Each of these interactions is a 3-tuple (S_i, A_i, S_{i+1}).

The following requirements can be placed on the model-learning process and its resulting model to ensure its applicability in assisting an agent during its decision-making process:

- **Accuracy:** First and foremost, forward models need to be accurate in their prediction of future states. This does not necessarily mean that the predicted future state needs to be correct, but that elements of the state necessary for the success of the current task need to be predicted as accurately as possible. This requirement will be the focus of evaluating the success of the learning process. For this purpose, statistical evaluation measures can be used. For example, predictions of deterministic processes can be evaluated using (weighted)

accuracy [110] or nondeterministic processes using Kullback-Leibler divergence [93]. How these measures will be adopted regarding the characteristics of evaluated models will be discussed in Section 4.10.

- **Processing Speed:** Keeping the computation time of the learned model as short as possible ensures that it can be applied multiple times during the simulation-based search procedure. Especially, in real-time scenarios, in which the reaction time of the agent is limited to a fixed time-span, a fast model is required to run a sufficient amount of simulations. Previous studies have shown that increasing the number of simulations has a positive impact on the agent's performance [49], especially UCT has proven to be converging to an optimal action-selection [163]. For this reason, when comparing two models with the same accuracy the model with the shorter processing time should be chosen to assure a higher number of rollouts. Similarly, the principle of Occam's razor may be applied to choose the simplest model among the models with the highest prediction accuracy.

- **Generalisation:** Two types of generalisation are considered. First and foremost, proposed forward model learning techniques should be applicable to a wide range of environments. Secondly, models trained to predict transitions of a specific environment should not only be able to predict already observed interactions with high accuracy, but also generalise well to unseen samples. This is especially important in case the training environment is not the same as the test environment. Here, the agent may face previously unseen situations to which it needs to respond appropriately. Machine learning problems such as overfitting and underfitting can apply here and need to be checked for by, e.g. cross-validation or the usage of a train-validation-test split [18].

- **Learning Speed:** Due to restrictions on the agent's response time, it may be necessary to learn or update forward models quickly. Even with a pre-trained model the agent may need to update its forward model in case newly observed interactions deviate too often from their predicted results. Especially, in the case of a dynamic environment, frequent

updates of the learned forward model seem inevitable. For this reason, keeping the training time of a forward model low ensures that the agent can frequently retrain its forward model of the environment.

- **Interpretability:** Not a strong requirement, but an interesting characteristic of forward models is their interpretability. A humanly interpretable forward model may be used to explain the agent's action-selection. In contrast to reinforcement learning-based systems, the agent does not only provide an expected value of a state (in form of the learned expected return) but may provide access to the search process and a sequence of hypothesised states. Comparing the expected outcome to the real outcome may explain why the agent chose a certain action. Furthermore, using an interpretable model allows users to compare the agent's learned forward model with their knowledge of the environment. This can help to improve the model building process.

4.2 End-to-End Forward Model

The end-to-end forward model represents the most basic class of forward models to be learned. Next to the introduction of this forward model type, comparisons of forward model learning and reinforcement learning will be made. The end-to-end forward model requires the agent to be able to differentiate observed states by a unique state id. The same applies to the agent's actions. While in reinforcement learning algorithms the agent learns a direct mapping of possible state-action pairs to the expected return, the latter is being replaced by the expected successor states.

The following subsections will provide a mathematical definition of end-to-end forward models (Subsection 4.2.1) and several extensions to take care of its apparent problems (Subsection 4.2.2).

4.2.1 Model Definition

During the learning phase, the agent needs to update its learned forward model based on previously observed transactions. In case the environment is deterministic, a transaction is identified by its state-action pair (S_t, A_t) and gets assigned the resulting state (S_{t+1}). In non-deterministic games, the

frequency of observed successor states needs to be stored to approximate the environment's probability distribution they are sampled from. The same process can be applied in case of deterministic games with a noisy state observation in which the element with the highest frequency is the most likely successor state.

When asked to make predictions about the result of a given state-action pair, the agent can query the database to see if similar situations occurred in the past. For this purpose, the database can be used as a lookup table. In case of a deterministic environment, the agent can report the results of previously observed interactions of the same state-action pair, or in a non-deterministic environment, it can sample from the set of all observed results by considering their frequency of occurrence. Therefore, the probability of a successor state S_{t+1} can be determined by

$$P(S_{t+1} \mid S_t, A_t) = \frac{\#(S_t, A_t, S_{t+1})}{\sum_{S' \in \mathcal{S}} \#(S_t, A_t, S'_{t+1})} \tag{4.2}$$

in which $\#(S_t, A_t, S_{t+1})$ is the number of previous interactions in which applying action A_t in state S_t yielded state S_{t+1}.

This process works in case the lookup table contains all relevant entries. However, if no such interaction has been observed, the agent still needs to be able to predict upcoming states with sufficient accuracy. A simplistic extension to Equation 4.2 would be to calculate the marginal probability of each state:

$$P(S_{t+1}) = \frac{\sum_{S \in \mathcal{S}, A \in \mathcal{A}} \#(S, A, S_{t+1})}{\sum_{S, S' \in \mathcal{S}, A \in \mathcal{A}} \#(S, A, S')} \tag{4.3}$$

Nevertheless, this prediction is very broad and assumes that the probability of the upcoming state is independent of the current state and the agent's action.

This estimation can be improved by the application of classification algorithms. Due to their ability to find frequently occurring patterns in a given training data set, they can be used to infer the outcome of new data points. The result is an approximation of the environment's model, which can be used to predict upcoming states. In the case of nondeterministic

games, probabilistic classifiers can be used. In contrast to deterministic classifiers, they estimate the probability of the current instance yielding any of the possible outputs.

4.2.2 Model Extensions

The previously described process of storing all observed interactions exhibits the same drawbacks as reinforcement learning algorithms. Depending on the size of the state and action space storing the resulting interaction table can become infeasible. While this approach suffices to model simple games (according to the size of their state and action space) this approach does not scale well with an increasing number of possible states or actions.

Instead of storing the result of each interaction individually, the content of the described interaction table can be compressed. If the number of states is known in advance, the complexity of the storage can be reduced by forcing a total order on the state space and storing the resulting state at its associated index. Nevertheless, these compression methods are limited in the reduction that they can achieve. Further compressions may be achievable due to exploitable dependencies in the transition function. For this reason, learning an end-to-end forward model can be feasible in case a large compression of the transition function is possible, e.g. the state transition function of Go or Chess can be explained by just a few rules despite the high complexity of both games' state spaces. Furthermore, the model size may be considerably smaller in case the agent does not learn the true model but only an approximation of it.

When applying forward model learning to games it will be the case that the agent can continuously add new observations to its training set. The problem with many classifiers is that they cannot adapt their current model to newly added instances. Instead, they need to be retrained using the updated training data set. As a result, the training data must be stored until the model is sufficiently trained. However, this process can take many iterations during which the training data set is continuously growing and may exceed memory limits. In this case, the application of online machine learning algorithms [61], also known as streaming machine learning algorithms, can help to overcome this problem. These can be updated with every newly observed data sample and do not need to store the whole training data set.

Even when storing all the previous interactions some of them may become outdated due to dynamic changes in the environment. Classifiers can cope with this scenario by using a dynamic weighting scheme. This way, new observations can be weighted higher than older ones. Likewise, a decay factor can be used to change the weight of training examples over time and allow the agent to remove training examples whose weight falls below a certain threshold. Similarly, the agent should approach new scenarios with quick rates of change in its forward model. This can be achieved by fixing a high learning rate which can be reduced as soon as the agent is able to sufficiently predict its environment. From there on, reducing the learning rate can make the agent's model resilient against noise in the observation of its environment.

As introduced in Section 4.1 the proposed forward model learning techniques require the environment to be a 1-st order Markov process. However, this requirement can be lifted by extending the models input from the current state and action to a sequence of previous states and actions. Given such a sequence of the n-previous states and actions, the model would theoretically be enabled to represent n-th order Markov processes.

$$\textit{n-th order fm}: \qquad \Big((S_{t-n+1}, \ldots, S_t), (A_{t-n+1}, \ldots, A_t)\Big) \longmapsto S_{t+1} \tag{4.4}$$

At the same time, this results in a drastic increase of the model's input and hypothesis space.

The size of the hypothesis space is what makes the extension to n-th order Markov processes infeasible in the context of the end-to-end forward model. Even for a 1-st order Markov process the number of possible models for a deterministic environment of n states and m actions is equal to n^{nm}. In the case of Chess which has a state-space complexity of $\approx 10^{47}$ and an estimated average branching factor of 35 this would result in:

$$10^{(10^{48} \cdot 35)} \approx 10^{10^{51}} = 10^{510}$$

While every new observation has the potential to drastically reduce the number of feasible models it is of high interest to analyse any potential of reducing the size of the hypothesis space.

4.3 Decomposed Forward Models

In the following, it will be shown how the forward model learning process can be improved by breaking down the overall learning task into a set of independent sub-problems. Since the end-to-end forward model does not make any assumptions on the state space, a model needs to be learned which maps to an arbitrary state-id. In case there is no inherent structure in the id assignment, this process becomes similar to brute-forcing the environment's forward model. More efficient compression of the model can be achieved in case there is an underlying structure of the target variable. While this may not be achievable for the state-id, it may be an inherent property of observed sensor values. Especially the observation of physical properties of the environment may uncover exploitable structures. For example, the change of an object's position is quite limited. At any given time, its next position is strongly dependent on the object's current position, its velocity, and applied forces by other objects. In case a state is defined by the observation of multiple sensor values, it may prove beneficial to predict changes of every sensor value independently and aggregating these predictions to infer the overall state. Reductions in complexity can be achieved since each sensor's model has a smaller hypothesis space and the observed values of a sensor may exhibit an exploitable structure.

A perfect decomposition becomes possible in case the change in a sensor's value is conditionally independent of changes in other sensor values when given knowledge of the current state and action:

$$\forall i, j \in 1..n : i \neq j \Rightarrow S_{t+1}^{(i)} \perp\!\!\!\perp S_{t+1}^{(j)} \mid S_t, A_t \tag{4.5}$$

If this property is satisfied, the forward model can be split into multiple independent submodels fm_i each modelling the transitions of a single sensor value based on the current state and the agent's action:

$$fm_i : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathcal{S} \qquad (S_t, A_t) \longmapsto S_{t+1}^{(i)} \tag{4.6}$$

Figure 4.1: Screenshot of Paku Paku[1], a free MS-DOS Pac-Man clone.

The original forward model can then be replaced by the aggregation of all submodels:

$$\begin{aligned} fm(S_t, A_t) &= (fm_1(S_t, A_t),\ fm_2(S_t, A_t),\ \ldots,\ fm_n(S_t, A_t)) \\ &= (S^{(1)}_{t+1},\ S^{(2)}_{t+1},\ \ldots,\ S^{(n)}_{t+1}) = S_{t+1} \end{aligned} \tag{4.7}$$

The following is a brief example to illustrate the proposed model's underlying principle and its impact on the model building process. Consider the game Pac-Man (see Figure 4.1). To be successful, the agent (yellow character) needs to traverse the maze while avoiding all ghosts (the pink, blue, orange, and red characters). While the agent's movement is only dependent on the chosen action, the behaviour of each ghost is dependent on its current position and its distance to the agent. Modelling the whole state transition means modelling the joint movement of all characters. Since each character can move into one out of four directions, a total of 4^5 state transitions are possible during every single state transition. By dividing the forward model into five submodels (one for the agent and one per ghost), the movement of each character can be modelled independently. In contrast to the original forward model, each submodel only needs to consider 4 possible outcomes. Combining the individual results yields the same state prediction while learning multiple submodels with reduced complexity in comparison to the overall forward model.

4.3.1 Automated Model Decomposition

A problem of the decomposed forward model can be its assumption of full independence among the predicted variables given knowledge of the current state and the agent's action. Since this assumption may be too strict, it is of interest to check which dependencies and independencies hold given a dataset of observed interactions. Detecting independencies among variables would enable the agent to filter irrelevant attributes before the forward model building process. Therefore, based on detected dependencies a conditionalised database can be constructed for each sensor value only containing attributes on which the future sensor value is dependent on.

The dependency analysis can be considered a structure-learning problem of a Bayesian belief net [51]. Let a Bayesian belief net be a directed acyclic graph denoted by $\mathcal{G}$ that encodes the dependence structure of a given probability distribution [23, 92]. The probability distribution $P(S_t, A_t S_{t+1})$ describes the likelihood of choosing action A_t in state S_t would lead to state S_{t+1} and is approximated given the history of previous interactions. Given a set of n observable sensor values the graph will consist of $2n + 1$ nodes in which n nodes represent the current state of each sensor, n nodes represent the future state of each sensor, and one node represents the agent's action. The graph's edges are used to encode dependencies and independencies of nodes in the graph.

Structure-learning algorithms try to identify a graph structure that matches the given probability distribution. While a complete graph can be used to encode any given probability distribution, structure-learning algorithms try to find independencies among observed variables to simplify the graph. Three categories of structure-learning algorithms can be identified, namely, score-based, constraint-based, and hybrid approaches [156].

Scoring-based Algorithms: Scoring-based approaches first generate a selection of candidate structures and select the most suitable based on a quality criterion. This quality criterion, e.g. the Bayesian information criterion [154], describes the correspondence between the implicit probability distribution of the structure and the underlying data set.

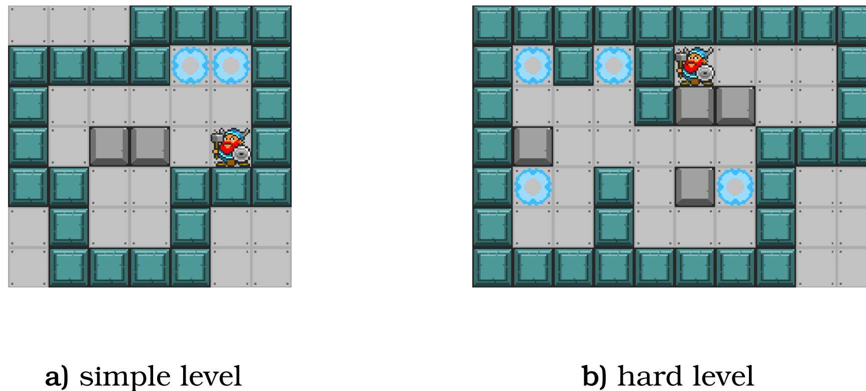
Since the number of possible structures super-exponentially depends on the number of nodes of the graph, a complete search can only be performed

in few cases. Therefore, the generation of candidate structures is usually limited to a local search. For example, the Hill-Climbing (*hc*) algorithm [40, 65] optimises an initial candidate structure step by step by adding, reversing, or removing edges. The best-rated structure will be further optimised during the next iteration until the search converges. An alternative search method is the tabu search (*tabu*), which maintains a list of previous changes [24]. In the course of the search, these can be undone to avoid local optima.

Constraint-based Algorithms: Constraint-based approaches use conditional independence tests to learn the dependency structure of attributes from the observed sample. The Grow Shrink (*gs*) algorithm [104] uses Markov blankets [103] to first determine an undirected graph structure. Thereafter, a directed graph is formed, by aligning edges, removing existing cycles, and propagating directions to undirected edges.

The semi-interleaved Hiton Parents and Children (*si-hiton-pc*) algorithm [3] first identifies local causal and markov-blanket relationships. Their combination results in an undirected candidate structure for the Bayesian belief graph. Similarly, the Max-Min Parents and Children (*mmpc*) algorithm [173] first generates an undirected graph structure by identifying all possible parents and children per node. Subsequently, scoring-based approaches can be used to determine the orientation of each edge. Both *mmpc* and *si-hiton-pc* should be particularly suitable for mapping the dependencies of large attribute sets due to their multi-level optimisation of the undirected graph structure. Therefore, these two algorithms could be particularly useful for complex state representations.

Hybrid Algorithms: Hybrid algorithms combine the two approaches presented above in a 2-phase procedure to efficiently find directional graph structures that fit a given data set. During the restriction phase, constraint-based algorithms are used to reduce the number of possible undirected candidate structures. Afterwards, edges are aligned using score-based algorithms in the subsequent maximisation phase. This general procedure is described in the *rsmax2* algorithm, in which the user can select any



a) simple level **b)** hard level

Figure 4.2: Two levels of the game Sokoban

combination of constraint-based and score-based algorithms [155, 156]. A representative of these algorithms is the Max-Min Hill Climbing (*mmhc*) algorithm [174] which combines the *mmpc* algorithm with the *hc* algorithm.

Structure-Learning for Games

The structure-learning process will be exemplified based on data retrieved from the game Sokoban. Sokoban is a classic grid-based puzzle game in which the player must push crates into designated locations. The player can move in four directions (up, down, left, and right). Moving into a box pushes it in the direction of movement, except the target position is blocked by either a wall or another box. However, the agent is not able to pull boxes. Therefore, pushing boxes into a corner yields a state in which the agent is unable to solve a level. Figure 4.2 shows two example levels of Sokoban with varying complexity.

Here, the agent perceives a game-state as a grid of tiles in which each tile uniquely describes if it contains either the player, a box, a wall, a target position, or if it is empty. Depending on the level's size the number of perceivable states can be quite high. An end-to-end forward model approach will not be efficient since there is no natural state-id assignment. For a grid of width w and height h, the agent can observe wh sensor values. Using a decomposed forward model a separate model needs to be learned for each of these sensors. Since the input of the model is quite complex, a pre-selection of the model's input can be recommended.

A data set of game interactions is built by letting a random agent play the simple level shown in Figure 4.2 for 5000 game steps. To get a diverse data set the level is reset after a maximum of 100 ticks or in case the agent finishes the level early. The data set consist of each tile's state before and after the interaction and the agent's action.

In the following, algorithms for learning the structure of directed belief graphs are compared, including the score-based algorithms *hc* and *tabu*, and the hybrid algorithms rsmax2 for any pair of the constraint-based algorithms *gc*, *mmpc*, and *si-hiton-pc* with any of the two scoring-based algorithms. During this process, multiple dependencies are blacklisted by providing a list of edges that cannot be included in the final belief net. Each edge starting at node u and pointing to node v that applies to one of the following rules is excluded:

- u and v both reference a tile's state before the interaction
- u and v both reference a tile's state after the interaction
- u references a tile's state after the interaction and v a tile's state before the interaction

The first two rules ensure the independence assumption of the decomposed forward model, while the third rule ensures that each dependency follows the chronological order.

A visual comparison of the resulting belief nets is shown in Figure 4.3. For easier visualisation, the nodes of the graph are shown at the position they indicate. Each tile is represented by a single node and edges from one state to the other indicate a dependency of the starting node's previous state and the target node's next state. Additionally, summary statistics of the structure-learning algorithms and their resulting graphs are presented in Table 4.1.

Resulting belief nets are compared based on their number of edges, number of structure evaluations during the optimisation process, and their final network score. For the latter, the Bayesian information criterion (BIC) is used.

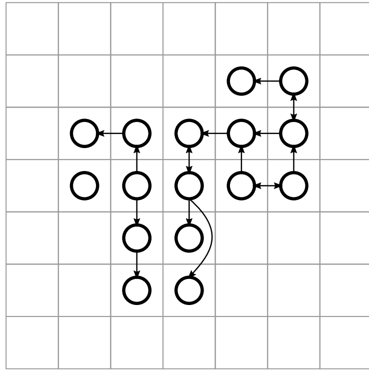
The evaluation shows that the scoring-based algorithms performed better in terms of BIC, but also tended to report a larger number of edges. While

Table 4.1: Summary of learned structures for the game Sokoban; #E: number of edges, #T: number of network tests during the learning process (hc/tabu), BIC: Bayesian information criterion

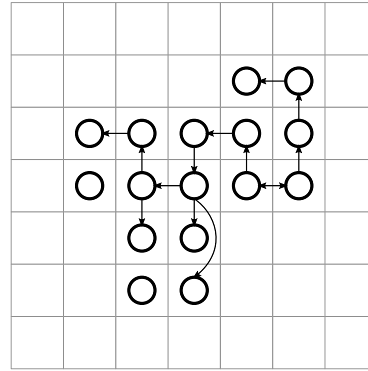
algorithm	*simple level*		
	#E	#T	BIC
hc	43	4301	-56034.66
tabu	43	1309	-56044.29
rsmax2 (gs, hc/tabu)	9	1652/1674	-76649.20
rsmax2 (mmpc, hc/tabu)	32	9183/9705	-57818.19
rsmax2 (si-hiton-pc, hc/tabu)	30	4508/4557	-57989.35

the hill-climbing algorithm resulted in a slightly better BIC, the tabu search needed less than a third of the evaluations. Both, the hill-climbing and the tabu search optimisation of edge directions resulted in the same graph for all hybrid algorithms but needed a different amount of graph evaluations. In contrast to only applying the scoring-based algorithms, the tabu search needed more evaluations than the hill-climbing algorithm. Learning the belief net's structure using the grow-shrink algorithm resulted in the worst performance among tested algorithms. Most nodes were considered to be independent, which may be due to the rarity of changes in the data set. Both, the *mmpc* and the *si-hiton-pc* algorithm performed better than the grow-shrink algorithm in terms of BIC. While the *mmpc* algorithm performed best among the hybrid-algorithms, it needed more than double the evaluations of the *si-hiton-pc* algorihm. Nevertheless, their results were slightly subpar to the results of the pure scoring-based algorithms.

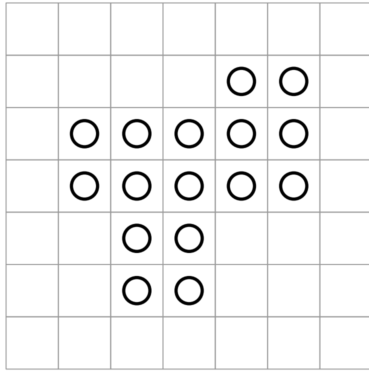
An interesting property of the reported graphs is that the strongest dependencies are reported among nodes that are close to each other. This perfectly represents the local dependence of Sokoban's forward model. Here, the character's and the boxes' movement can perfectly be described by taking neighbouring tiles into account.



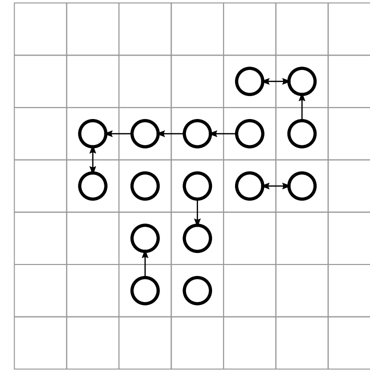
a) Hill Climbing



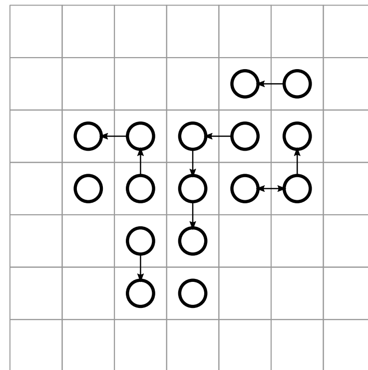
b) Tabu Search



c) Grow-Shrink + Hill Climbing/Tabu Search



d) Max Min Parent and Children + Hill Climbing/Tabu Search



e) Semi-interleaved Hiton Parents and Children + Hill Climbing/Tabu Search

Figure 4.3: Comparison of learned dependence structures. The position of each node encodes the state of each moveable tile in the simple level (cf. Figure 4.2a). A directed edge from node $u \to v$, $u \neq v$ indicates a dependence of their associated sensor values S_t^u and S_{t+1}^v. Edges of type $S_t^u \to S_{t+1}^u$ and $A_t \to S_{t+1}^u$ are ommitted for a clearer presentation.

4.4 Local Forward Models

In the previous sections, it was demonstrated how a model can be decomposed into multiple sub-models. To reduce the computational effort a pre-filtering of features can be applied before each model's building process. However, the detection of sensor dependencies requires a large dataset of previous interactions to be accurate and needs to be separately done for each sensor value. This section will show how the computational complexity of the model building and the feature selection process can be reduced in case the agent is aware of a semantic arrangement of sensor values.

Such a semantic arrangement can be observed in the visual output of a game. Here, the agent can observe the colour value of each pixel and its position. Pixels that are closer to each other often correspond to the representation of the same object. Moving the object results in a position change of all corresponding pixels. However, the meaning of these pixels does not change when moved to another position since they still represent the same object. Similarly, grid-based games such as Chess or Sokoban let the agent observe every cell of the board. Moving a piece from one cell to another changes the observed value of associated grid cells, but does not change the meaning of the piece itself. Its possible action space remains the same despite the position change.

In the presented experimental evaluation of structure-learning algorithms, it was observed that the transition of a single grid cell can often be modelled without knowledge of the whole state. Instead, in the case of the analysed grid-like structures, it was observed that a small distance between objects is a strong indicator of dependence. The observed local dependence among objects and the unchanged meaning of repositioned sensor values form the main motivation of local forward models.

In the following, two main assumptions need to be satisfied for the applicability of local forward models. First, it is assumed that sensor values are arranged in a graph-like structure on which a neighbourhood relation among sensor values can be defined. This will allow differentiating local and global interactions between objects. Local interactions describe dependencies between two neighbouring objects' sensor values, whereas global interactions describe dependencies between any two non-neighbouring objects' sensor

values. Local forward models focus on the former by only modelling local interactions and ignoring global interactions. The second assumption of local forward models is that changes of each sensor value can be described independently of its global position by only taking its current value, the value of neighbouring sensors, and the player's action into account.

In the following Subsection 4.4.1, local neighbourhoods will be defined. The process for learning local transition functions will be described in Subsection 4.4.2.

4.4.1 Local Neighbourhoods

A local forward model describes the changes in each sensor value by

$$fm_i : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathcal{S} \qquad \left(N(S_t^{(i)}), A_t\right) \longmapsto S_{t+1}^{(i)} \tag{4.8}$$

whereas $N(S_t^{(i)})$ is the local neighbourhood of sensor value i at time t. Given a distance threshold ε and a distance metric $d(x, y)$ the local neighbourhood is given by

$$N(S_t^{(i)}) = \left\{S_t^{(j)} \mid d(S_t^{(i)}, S_t^{(j)}) \leq \varepsilon, \quad j \in 1, \ \ldots, \ n\right\} \tag{4.9}$$

Therefore, it includes all sensor values that are closer than a given threshold. However, the definition of this neighbourhood largely depends on the given structure and its associated distance metric.

For simplicity, it will be assumed that all sensor values of a state's representation are arranged in a grid-like structure, e.g. grid-based and pixel-based state representations. Hence, a state can be represented as a set of tiles or pixels arranged in a grid in which $T(x, y)$ specifies the tile at position (x, y) on a grid in euclidean space. For each cell, a model will be built which can predict the future state of the cell based on its current state and its local neighbourhood.

Let the local state transition function $f_{x,y}$ be given by

$$fm_{x,y} : \left(N(x, y)_t, A_t\right) \longmapsto T(x, y)_{t+1} \tag{4.10}$$

for which $N(x, y)_t$ describes the local neighbourhood of cell (x, y) at time t and A_t the agent's current action. The Minkowski distance[149] can be used

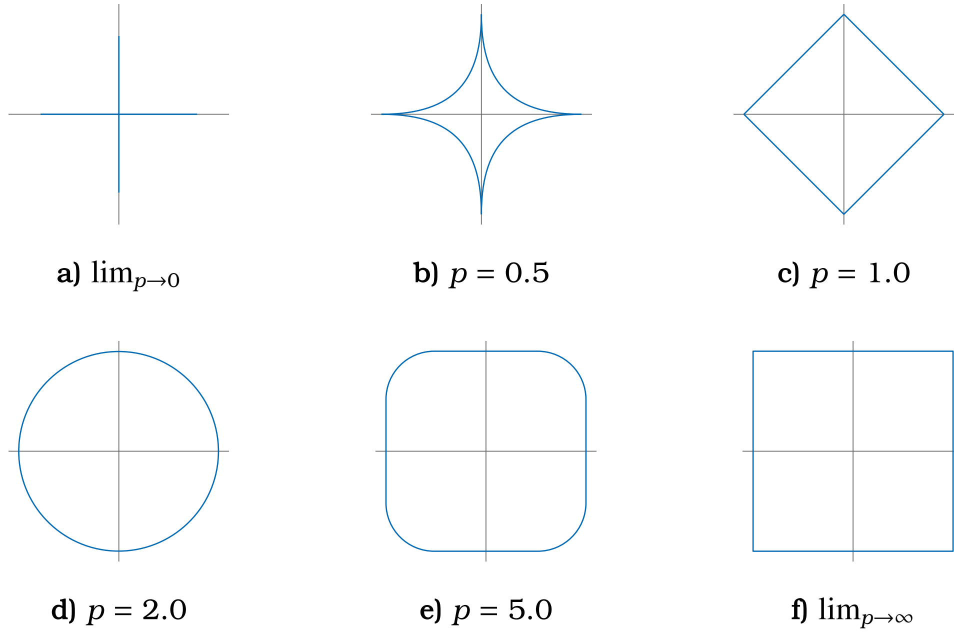


Figure 4.4: Unit circles (blue lines) of the Minkowski distance for varying values of p.

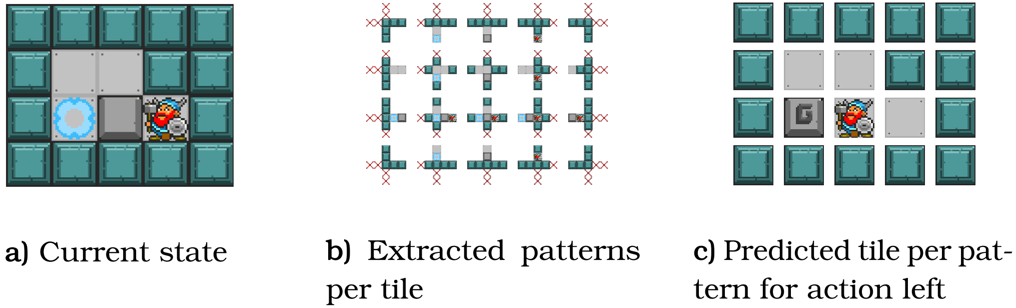
to measure the distance of two points in euclidean space $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n) \in \mathbb{N}^n$, which is defined by

$$D(X, Y) = \left(\sum_{i=1}^{n} |x_i, y_i|^p \right)^{\frac{1}{p}} \tag{4.11}$$

for $p \geq 1$. The Minkowski distance with $p = 1$ corresponds to the Manhatten distance, $p = 2$ to the Euclidean distance, and the limiting case of p reaching infinity to the Chebyshev distance. For $0 \leq p < 1$ the Minkowski distance does not fulfil the triangle inequality, and therefore, is not a distance metric. Nevertheless, it can still be used to define the local neighbourhood of a cell since the neighbourhood relation does not need to be transitive in this application. The resulting shape of a cell's local neighbourhood is shown in Figure 4.4 for various values of p.

4.4.2 Learning Local Transition Functions

Given the definition of a local neighbourhood, the local transition function can be learned based on previously observed cell transitions. Similar to



a) Current state b) Extracted patterns per tile c) Predicted tile per pattern for action left

Figure 4.5: Visualising steps of the local forward model's prediction process.

previously discussed forward model learning techniques, learning the local transition function is equal to learning a classifier, which can predict the upcoming sensor values.

When assuming that the semantic of a sensor value is independent of its represented position in the grid, it suffices to learn a single model that applies to all grid cells. Additionally, the data collection process becomes more efficient since instead of recording a single game-state transition per game tick, a separate transition can be recorded for each cell in the grid. Depending on the grid size this can result in a massive amount of training samples by just observing a single game-tick. Hence, this technique might be able to considerably lower the number of interactions to be observed during the training process. The extraction process for local neighbourhood patterns is shown in Figure 4.5.

While in an end-to-end forward model, the model is called only once to predict the next game-state, a local forward model needs to be applied to each cell separately. However, this process can be completely parallelised since the upcoming cell states are assumed to be independent of each other. To do so, the agent extracts the local neighbourhoods per cell and applies the learned local forward model to predict each cell's next value. In case the trained classifier allows the use of batch-processing this can further speed up the prediction process considerably.

4.5 Object-Based Forward Models

Previously discussed forward models were either based on the state-id or the analysis of a state's sensor values. Such low-level based modelling allows the learning procedure to be applied to a wide range of scenarios. However, incorporating background information on the state description may allow to further improve the model learning process. Such high-level information could be provided in the form of, e.g. entity-based sensor groupings, object types, collision events, etc. In the following, an object-based forward model learning procedure will be proposed, which can incorporate such high-level information in the prediction of future states. Specifically, it will be discussed how an object-based representation of the environment can be used to reduce the number of models to be learned and, therefore, speed-up the learning process.

The object-based forward model is motivated by the object-based state representation of the GVGAI framework. For each visible sprite in the current game-state, the GVGAI framework provides the agent information on the sprite's current position, rotation, orientation, and multiple other attributes. An important aspect of this representation is that each sprite has a unique id which can be used for tracking the sprite over multiple time steps to allow the deduction of its movement over time. Newly spawned objects or destroyed objects can be detected by their id. While such high-level information in the state representation may not be provided by the environment, the agent may be able to construct its own representation by preprocessing a visual representation of its environment. Techniques for doing so will be discussed in Subsection 4.5.1.

In the following, it will be assumed that next to the state description in the form of its sensor values, the agent is aware of an entity-based partitioning of these sensors. A partition may describe the sets of sensor values $1, \dots, m$ that belong to a common object in the environment such as, e.g. the agent's avatar or a non-player character.

$$\begin{aligned} \mathcal{S} &= (S^{(1)}, S^{(2)}, \dots, S^{(n)}) \\ &= (\underbrace{S^{(1,1)}, \dots, S^{(1,i)}}_{\text{Object 1}}, \underbrace{S^{(2,1)}, \dots, S^{(2,j)}}_{\text{Object 2}}, \dots, \underbrace{S^{(m,1)}, \dots, S^{(m,k)}}_{\text{Object m}}) \end{aligned} \tag{4.12}$$

This information can either be provided by the environment itself or may be extracted by the agent from the state observation. Similar to a decomposed forward model, the object-based forward model assumes that the behaviour of each object can be modelled independently of other objects. Therefore, the forward model will be split into multiple submodels fm_i, whereas each submodel describes changes of object i's associated sensor values $(S_{t+1}^{(i,1)}, \dots, S_{t+1}^{(i,k)})$:

$$fm_i : \left((S_t^{(i,1)}, \dots, S_t^{(i,k)}), A_t\right) \longmapsto (S_{t+1}^{(i,1)}, \dots, S_{t+1}^{(i,k)}) \tag{4.13}$$

This object-submodel may further be decomposed into multiple forward models, whereas each sensor-model $fm_{i,j}$ predicts changes of the j-th sensor value $S^{(i,j)}$ of object i:

$$fm_{i,j} : \left((S_t^{(i,1)}, \dots, S_t^{(i,k)}), A_t\right) \longmapsto S_{t+1}^{(i,j)} \tag{4.14}$$

The original forward model can then be replaced by the aggregation of all objects submodels or their sensor submodels:

$$\begin{aligned} fm(S_t, A_t) &= (fm_1(S_t, A_t),\ \dots,\ fm_n(S_t, A_t)) \\ &= ((fm_{1,1}((S_t^{(1,1)}, \dots, S_t^{(1,k)}), A_t), \dots, fm_{m,k}((S_t^{(m,1)}, \dots, S_t^{(m,k')}), A_t))) \\ &= (S_{t+1}^{(1)},\ S_{t+1}^{(2)},\ \dots,\ S_{t+1}^{(n)}) = S_{t+1} \end{aligned} \tag{4.15}$$

Further reductions of the total model size can be achieved in case objects share similar forward models. This could happen in case each object belongs to a certain object type. Each object type can further have multiple instances, e.g. groups of enemies that behave the same. In case the environment provides us with information on an object's type or the agent is able to recognise the object's type based on its associated sensor values or its behaviour, object-based models of similar object types can be merged. This reduces the number of models to be learned. At the same time, it increases the number of training examples per model, since each instance provides unique observations for the forward model's training data set.

The feasibility of this approach will shortly be explained based on the game "alien" provided by the GVGAI-framework (see Figure 4.6). Here, the

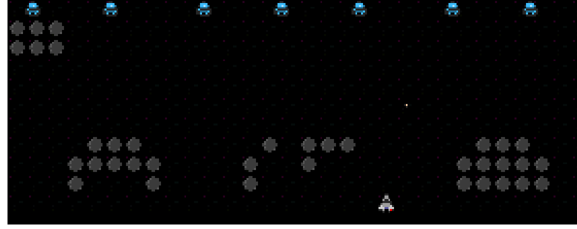


Figure 4.6: Gamestate of the game “aliens” from the GVGAI framework. Four types of entities: top: alien spaceships, middle: player shot and boulders, bottom: player spaceship

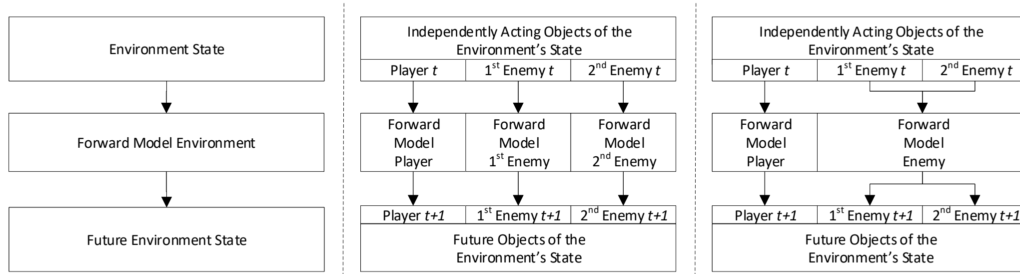


Figure 4.7: Comparison of forward model architectures: (left) end-to-end forward model; (middle) object-based forward model consisting of one sub-model for each object of the environment; (right) object-based forward model consisting of one sub-model for each type of objects

player controls a spaceship at the bottom of the screen, which can fly left and right and shoot a bullet upwards. The agent's task is to destroy alien spaceships which spawn at the top of the screen and traverse the screen from left to right. When they reached the end of a row they get down on the next one and reverse their horizontal direction. The player loses the game in case an alien reaches the down-most row.

Overall, the game consists of three moving object types: the player's avatar, shot bullets, and alien spaceships. Learning a separate model for each of the alien spaceships or each of the player's bullets will result in an unnecessary amount of models since the behaviour of instances of the same object type will be the same. In return, this means observations of all aliens can be used to train the agent's model of an alien spaceship. Therefore, during each tick, every alien provides a separate training example. The final model architecture is shown in Figure 4.7.

In the proposed model architecture it was assumed that each object can be modelled independently of other objects. In case objects are able to influence each other an object's model will need to access the sensor values of the object being modelled as well as the sensor values of other objects. Since the number of objects may change while playing the game the number of inputs per model would change as well. This can be avoided by not providing the object model with the sensor values of other objects but with calculated summary statistics. These may include the number of observed objects, their type, or even higher-level information like their average distance. This feature creation process may be adapted for the task at hand to ease the agent's learning process.

Similar to local forward models the object-based forward model has two major benefits. First, the proposed forward model is independent of the number of instances per object which means that the total number of observed sensor values can vary with each game tick as long as every object type is represented in at least one forward model. Second, multiple instances per object type can be used to train the same model which yields more training examples per model than training a separate model per object.

4.5.1 Generating Object-based State Representations

In the previous example, it was assumed that the segmentation of the state into multiple objects is provided by the state representation. Since this does not need to be the case, it will shortly be discussed how object-type information may be extracted from a visual state representation. This subsection aims to be a short overview of computer vision algorithms and how they can benefit this process.

For the purpose of creating an object-based representation, supervised and unsupervised algorithms can be used. Supervised object detection algorithms need a set of labelled examples to either train a model which identifies an object by its typical features, or to compute the similarity between extracted image patches and known labelled examples. In contrast, unsupervised algorithms may extract and cluster image patches to identify frequently occurring patterns. In both cases, the screen representation can later be replaced by the set of identified objects or the closest matching pattern per patch.

In case an object's representation is known, template matching can be an efficient method for the detection of objects. Since many 2-dimensional games are based on sprite sheets (e.g., character animations) and tilesets (e.g., non-moving objects), this information may be available during the development of an agent. In this case, template matching algorithms allow the agent to detect objects in the visual scene and describe them by their position, their bounding box, and assign a unique object id. Classical template-matching algorithms determine the difference between an image patch and a given template. This process can be improved by instead using a feature-based representation of provided templates [13, 179]. Depending on the used features, given templates can be reliably detected despite differences in background graphics, rotation, and colour variances. In deep learning-based template matching, these features are learned from labelled training examples. Deep-learning based approaches have proven to be effective in a wide range of image-classification tasks [91, 145]. Those pre-trained object detection networks may be applicable to the game's graphical output in case real-world objects are represented in the environment. For this purpose, the network weights of a pre-trained network are tuned for the current classification task by providing new training examples. Given a set of input images that represent similar content such as multiple frames of a game scene, co-segmentation algorithms can be used to extract commonly occurring templates. Hence, 2-dimensional or 3-dimensional scenes can be processed to detect fore- and background objects.

Object tracking algorithms can further be used to track these instances over time [188]. This becomes necessary to record each object's change in position or orientation and use this information for training a forward model. Furthermore, overlapping boundary boxes can be used to report collision among objects for a more feature-rich representation.

An unsupervised object-detection algorithm has been used by Chen et al. [37] to cluster game objects in the visual state representation. Their proposed algorithm is restricted to tile-based games in which observed tiles are extracted and clustered. After an initial learning phase newly observed tiles are assigned to the cluster with the most similar instances. The constructed object-based representation has shown to benefit the reinforcement learning-based agent in terms of a reduced learning time and model size.

4.6 Comparison of Proposed Learning Methods

In this section, the four proposed forward model learning methods will be compared based on the properties described Section 4.1. A schematic overview of the proposed methods is shown in Figure 4.8. and results of this comparison are summarised in Table 4.2.

The most general model in terms of applicability is the end-to-end forward model. This model can be applied in all scenarios in which the agent can identify a state based on its observable properties. This general applicability, however, leads to problems regarding the size of the hypothesis space and its associated number of necessary training examples. Hence, the flexibility of this approach comes with the drawback of slow learning speed, high memory consumption, and almost no generalisability regarding previously unseen examples.

The other methods try to overcome these problems by introducing assumptions on the state-space to reduce the size of the hypothesis space. The decomposed model framework and the proposed automated model decomposition can both be applied without assumptions on the underlying state representation. In contrast, the local forward model approach is based on semantic relationships of neighbouring sensor values. The object-based forward model requires the agent to be aware of independently acting components in the state-representation. The process becomes more efficient in case these components share common properties. Both methods considerably reduce the number of input variables per model and the overall number of submodels to be learned. However, they also make strong assumptions on the state representation which may not be fulfilled in the given environment but could be fulfilled by preprocessing a phase.

Similar to the generalisability of the learning process to unknown environments, the transfer of an existing model to previously unobserved states is an interesting characteristic of forward models in the context of general game learning. In this form of transfer learning, the agent can be confronted with new situations such as an increased level size or new components in the environment. In such a case, it would be desirable to adopt knowledge from previously observed levels to shorten the time it takes the agent to adapt to the new situation.

Both, end-to-end forward models and decomposed forward models, would need to be retrained in case the environment introduces new states or a differing number of sensor values. In contrast, the local forward model and the object-based forward model can reuse the knowledge of previously observed transactions if the new level uses a state representation analogous to the levels at which the agent was trained. The transition function learned by local forward models is independent of a tile's absolute position. Because of this, the transition function can be applied to every tile in the state representation even if the total number of tiles varies in the newly observed state. Similarly, the object-based forward model can be applied as long as objects of new levels are similar to previously observed objects. The use of summary statistics as inputs of each model allows the model to be applied in case of a varying number of objects in the environment.

The interpretability of the model depends on the model's structure, the classifier being used, and the complexity of the environment. While the latter cannot be influenced by the agent, the first two directly depend on the developer's choices during the modelling process. As discussed above, the complexity of the end-to-end forward model is the highest since it needs to cover all possible state transitions in a single model. The other models are likely to be more interpretable since they are each focusing on a single aspect of the state representation. While the decomposed forward model offers a submodel for each sensor value, the object-based forward model describes updates of a whole object. The latter can be beneficial in cases in which an object's sensor values are dependent on each other. In case the model is able to distinguish groups of similar objects, the resulting reduction of models to be trained can enhance the interpretability of the overall forward model. Similarly, the reduction in the number of submodels for the local forward model, which applies the same model to each observable tile, can lead to an easily interpretable forward model.

Next to the structure of the forward model, the choice of the classifier has a direct influence on the model's interpretability. Especially classifiers that are represented in the form of rules or simple decision trees have often proven to be humanly interpretable [19]. Prototype-based classifiers such as k-nearest neighbour justify the decision based on previously observed examples. Depending on the similarity measure being used and the number

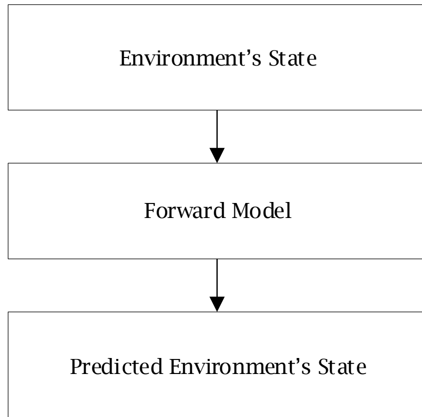
Table 4.2: Qualitative comparison of proposed forward models architectures; (+) well suited, (~) neutral, (—) poorly suited

Forward Model	#Models	Model Complexity	Interpret-ability	Transfer across levels
End-To-End	+	—	—	—
Decomposed	—	~	~	—
Local	+	+	+	+
Object-based	~	~	+	+

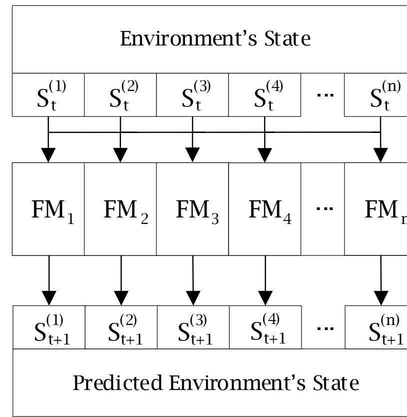
of considered neighbours these classifiers can be interpretable by humans to some degree. Similarly, ensemble-based methods can become hard to interpret in case they are aggregating the results of many classifiers.

The usage of black-box classifiers such as neural networks or support vector machines can be more problematic. Since the internal representation of these classifiers does not allow direct interpretation, reverse engineering methods need to be used for the extraction of interpretable knowledge. Model-specific techniques such as the extraction of rules for deep neural networks [76] and the visualisations of neural network layers [125] have both proven useful in getting insights into the neural network's classification process. On the contrary, model-agnostic frameworks measure the influence of input parameter on the outcome of the classifier, e.g. sensitivity analysis [41] or feature importance [81].

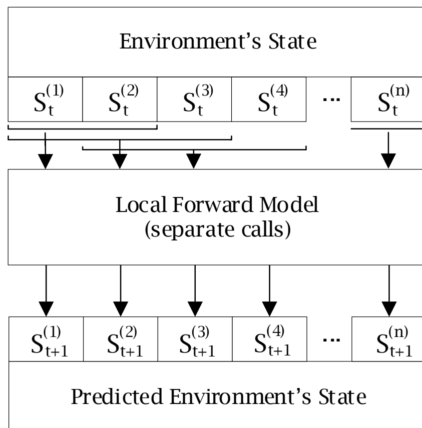
Even if the classification process cannot be regarded as humanly interpretable, analysing the output of the forward model can be valuable in verifying the accuracy of the model. The visualisation of the result can prove useful in understanding the predictions made by the forward model. Since the local forward model is directly associated with tiles that may have a known graphical representation, a visualisation of the state becomes possible. The same applies to the other models in case it is known how predicted parameters relate to the graphical output of the game. Especially, in case an object-based forward model was trained using data of visual pre-processing techniques, the visual representation of an object may be known. Finally, the end-to-end forward model directly maps to the agent's state observation. Therefore, the model's result can be visually inspected in case the model was trained on a visual state observation.



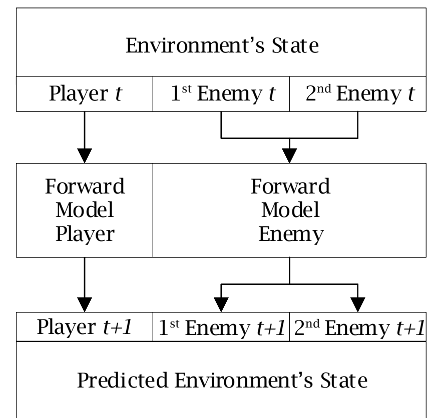
a) End-to-End Forward Model



b) Decomposed Forward Model



c) Local Forward Model



d) Object-based Forward Model

Figure 4.8: Visual comparison of proposed forward model architectures

4.7 Agent Model

Simulation-based search agents make use of the environment's true forward model to analyse the outcome of an action. Since a trained forward model serves as a replacement for the environment's true forward model, a similar process can be applied. To differentiate the use of these different forward models, a search using a trained forward model will be called a prediction-based search.

In case the trained forward model should remain unchanged over time, the search process resembles a simulation-based search. Measuring the model's accuracy would allow for the application of more specialised search methods which compensate for the agent's confidence in the predictions.

If the agent is allowed to update the trained forward model, it needs to keep track of its recent interactions with the environment. Every observed interaction could consist of previously unobserved patterns which can be added to the model's training data. Retraining the forward model would be done by updating the classifier. To keep track of the model's accuracy, the agent can compare past predictions with the observed outcome after executing an action. It is recommended to update the classifier after an error has been observed to avoid repeating the same mistake. This feedback loop is key to continuously improving the agent's model of its environment and is shown in Figure 4.9.

The trained forward model allows the agent to predict upcoming states and, therefore, the outcome of the agent's actions. Given a task description the agent would be able to detect states in which the task is completed, thus, searching for appropriate action sequences becomes possible.

In case the search process requires an evaluation of the value of intermediate game states, it can be beneficial to also learn a reward model. The reward can be incorporated in previously proposed forward models by representing it as an additional sensor value. Alternatively, reinforcement learning algorithms can be used to learn the value of a state while the agent is playing the game. Therefore, prediction-based search algorithms do not need to replace reinforcement learning approaches but can be used in con-

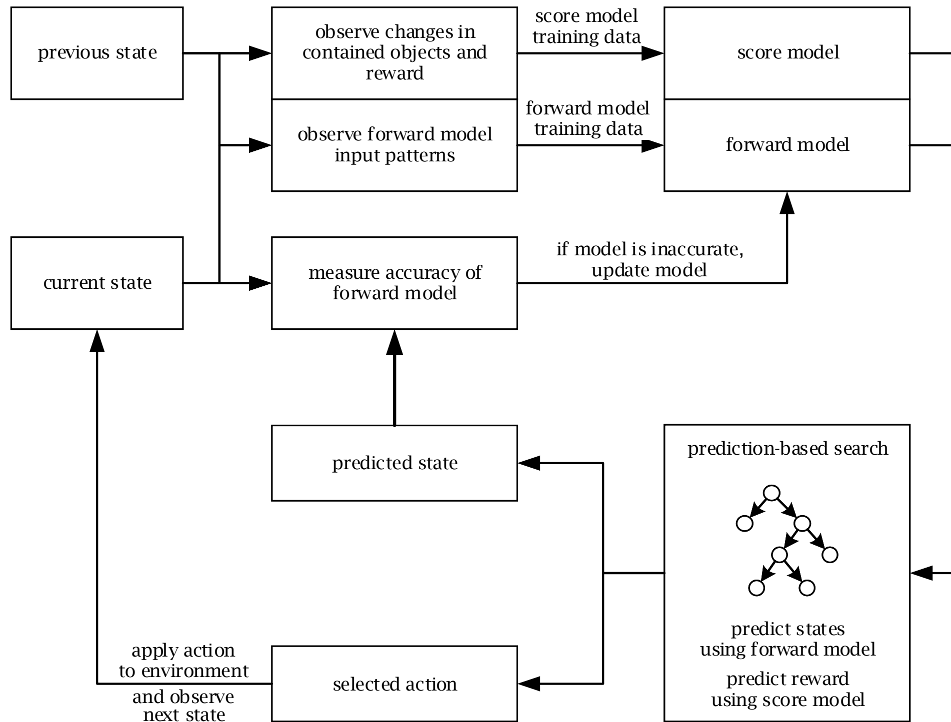


Figure 4.9: Prediction-based search including continuous model updates.

junction with them. This also creates the possibility of using a reinforcement learning algorithm in a simulated environment that is updated according to the agent's forward model, which allows for a safe learning environment.

In the context of this thesis, a simple score model will be used to predict the reward of a state transition. This score model will be implemented using a linear regression which maps the differences of the current state and the upcoming state to a reward value. The difference vector contains entries for every observable tile or object type in the state representation and consists of their number of occurrences, the number of destroyed individuals, and the number of newly created individuals per tile or object type. This model was chosen since it can be used for all proposed forward models and requires minimal additions to the prediction-based search.

4.8 Results of Previous Work

A preliminary version of the decomposed forward model was proposed by Daan Appeldoorn and I in the paper called “Forward Model Approximation for General Video Game Learning” [47]. In this study, the agent was tasked to play games of the GVGAI framework without having access to the environment’s forward model. Instead, the agent needs to base its decisions on predictions of the avatar’s movement, scored points, and the game’s termination conditions. These predictions were made using hierarchical knowledge bases [6] that were trained to predict the upcoming state based on one minute game-play of a random agent. After the training phase, the agent analyses the current state and applies the most specific association rule to predict the next state. The proposed agent used either BFS or MCTS to play 10 different games. In the evaluation the agent’s game-playing performance was compared to agents of the 2017 competition’s learning track. The proposed agent was able to overall outperform these agents and was the best performing agent in 5 out of 10 games. Nevertheless, the model’s prediction capabilities were still limited.

The aim of improving the model’s prediction accuracy and extending its applicability to all game elements lead to the development of the decomposed forward model. A more flexible prediction framework based on an ensemble of complementary decision trees was proposed in the follow-up paper [56]. Here, the changes in each sensor value were predicted independently using a decision tree classifier. The proposed technique resulted in prediction accuracies between 60 and 90 % per game. However, the number of models to be trained was dependent on the number of observable sensor values and has shown to be infeasible for games with larger game-states.

To reduce the number of input parameters per model the application of a dependency analysis was first studied in my paper “Detecting Sensor Dependencies for Building Complementary Model Ensembles” [51]. A qualitative analysis on the two games “aliens” and “butterflies” of the GVGAI framework has shown that stochastic independencies among variables can often be detected after just a few interactions (~100-1000). Results of this

evaluation suggested that in structured state observations, such as a grid of observed tiles, the next state of a tile can often be predicted based on the current state of neighbouring tiles.

The local forward model was first used in our paper “A Local Approach to Forward Model Learning - Results on the Game of Life Game” [98]. Here, the model has been tested in the context of Conway’s Game of Life, a simulated 2-dimensional grid world based on a cellular automaton. In the beginning, each cell of the grid is randomly initialised so that it is either alive or dead. A simple set of rules determines the future state of each cell depending on the current state of a cell and the number of living cells in its vicinity. Despite its name, Conway’s Game of Life is not a game in the conventional sense, since the player cannot interact with the environment after the simulation has been started. In our version of the Game of Life, the agent could activate or deactivate one cell per tick. The reward of the agent was determined depending on the number of living cells per tick. Both, the grid-based state representation and the update rule, clearly fulfil all assumptions of the local forward model. Thanks to the simplicity of the Game of Life’s forward model, it is possible to determine all local patterns and their results. The evaluation showed that, after a small number of observed patterns, the model was able to generalise its prediction well to unobserved patterns. Since the observation of a single interaction results in a large number of observed patterns, the agent quickly learned a suitable model to keep large parts of the population alive, as opposed to a random agent for which the population died out quickly. This work also attempted to apply the local forward model to games of the GVGAI framework without much success.

In addition to our experiments on the Game of Life, the local forward model has also been tested in the context of the game Sokoban [52]. Sokoban (published by Thinking Rabbit in 1982) is a classic grid-based puzzle game in which the player must push a determined number of crates into designated locations to complete each level. The player can move in four directions (up, down, left, and right) and push boxes in the direction of travel. In contrast, pulling boxes is not possible. Experiments have shown that the agent is able to learn how to play a single level after just a few attempts. Additionally, the agent’s ability of transfer learning across multiple levels has been tested by training the model on 10 levels and evaluated the agent’s

game-playing performance on 10 new levels. Despite being confronted by previously unseen game-states the agent was able to predict the outcome of its actions well enough to outperform non-learning algorithms.

Next to the prediction of the upcoming state, it has also been tested if it is possible to predict the agent's reward and the game's termination criteria. For this purpose, the use of association rule mining algorithms has been proposed in the paper "Association Rule Mining for Unknown Video Games" [54]. This process performed well when predicting common state transitions but failed to represent rare events. Especially association rules describing the termination condition often failed to reach the minimum support threshold. Lowering the threshold until these rules are reported has shown to be infeasible due to a large number of uninteresting rules being reported as well. This problem was solved by the proposal of a 2-step process. First, association rules that predict the game's termination were extracted based on a data set only containing the final state transition of each attempt. Secondly, association rules that were incorrectly activated in any of the previously observed interactions were filtered. The proposed algorithm was tested on games of the GVGAI framework and showed to be capable of returning correct and humanly interpretable rules.

4.9 Evaluation Setting

The GVGAI framework provides a unified interface for more than 100 games. The following experiments will be based on the Python client[2] of the GVGAI single-player learning track. This client allows the agent to access a visual representation or a grid-based representation of the current game-state. Access to the object-based state representation sent by the server was assured through slight modifications of the framework. In the following, the grid-based representation, as well as the object-based representation, will be used to train respective forward models.

Many of the GVGAI games use grid-based physics. However, the grid resolution used in the environment model does not always match the grid resolution of the state observation. This can lead to game components being reported in multiple cells at the same time. In the following evaluation, only

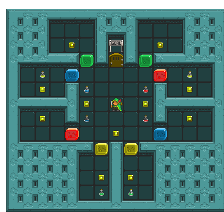
[2] `https://github.com/rubenrtorrado/GVGAI_GYM`

games in which each game object is only represented once per time step will be used. Additionally, these games will be required to offer at least 5 levels to assure that the agent is able to observe a diverse set of game-states per game. The following paragraphs shortly introduce three of these games which are shown in Figure 4.10. Descriptions and visualisations of the remaining games can be found in Section A.1.

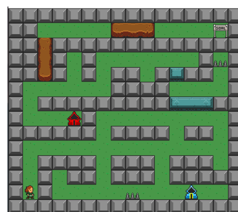
chipschallenge: Each level includes up to four coloured blocks which cannot be passed unless the agent collected a potion of the same colour. The agent can increase its score by collecting coins. A door, which is blocking the path to the final exit, is destroyed in case the agent collected at least 11 coins when touching the door. The game is won in case the agent reaches the final exit and lost in case the agent walks on a fire or water tile. However, dying can be avoided by first collecting boots of the same colour. Chipschallenge is one of the more complex games using larger levels and requiring the agent to manage multiple resources.

labyrinthdual: The agent represented by the small man in the bottom left corner needs to traverse the labyrinth to reach the goalpost. In case the agent touches the blue or the red house, the agent's colour changes accordingly allowing him to pass blocks of the same colour. Later levels allow the agent to get stuck in case it changes its colour in an inescapable position.

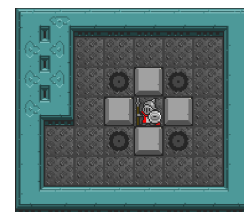
realsokoban: This game mimics the original Sokoban game in which the agent is tasked to push all the boxes on the tiles marked by a circle. However, boxes cannot be pulled. This can hinder the agent from finishing a level since the box may not be movable from any direction.



a) chipschallenge



b) labyrinthdual



c) realsokoban

Figure 4.10: Three games of the GVGAI framework

4.10 Evaluation of the Prediction Accuracy

In this first evaluation, the trained forward models' accuracy of predicting an upcoming state or its components will be tested. Since the prediction target of the local forward model differs from the object-based forward model both methods will be evaluated separately in the following subsections.

4.10.1 Accuracy of Local Forward Models

In this evaluation, it will be tested if the proposed local forward model approach is able to learn an accurate model of the environment. The influence of multiple parameters will be studied and it will be discussed how the best model can be identified.

In this work, three types of local neighbourhoods will be considered, namely the cross, the diamond, and the square pattern (cf. 4.11).

$$\begin{aligned} N_{cross}(x,y) = \{T(x+i,y) \mid 0 \leq |i| \leq span\} \cup \\ \{T(x,y+j) \mid 0 \leq |j| \leq span\} \end{aligned} \tag{4.16}$$

$$\begin{aligned} N_{diamond}(x,y) = \{T(x+i,y+j) \mid 0 \leq |i| \leq span,\ 0 \leq |j| \leq span \\ |i| + |j| <= span\} \end{aligned} \tag{4.17}$$

$$N_{square}(x,y) = \{T(x+i,y+j) \mid 0 \leq |i| \leq span,\ 0 \leq |j| \leq span\} \tag{4.18}$$

Various spans will be tested to investigate the influence of the number of input variables and unique training examples on the classifier's training process and result.

For the evaluation of local forward model learning the first step will be to find suitable classification algorithms and their parameter settings. For this purpose, a replay data set of a randomly playing agent will be generated. Here, all 5 levels were played 10 times for 200 ticks each while recording the observed state in each time-step. Note, that some replays are shorter since the agent loses the game before the tick threshold is reached.

In the following, the influence of the neighbourhood shape, its radius, the choice of the classification algorithm, and the algorithms' parameter settings

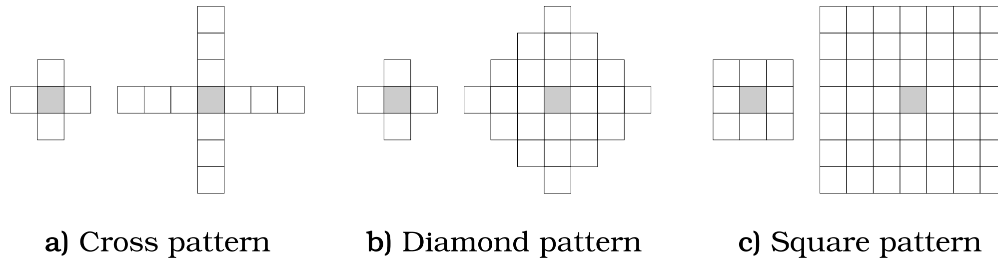


Figure 4.11: Local neighbourhood patterns encoding the local neighbourhood of the centre tile for span sizes 1 and 3

will be studied. Similar to the previous evaluation 3 neighbourhood shapes are tested. For each of these shapes the neighbourhood radii 1, 2 and 3 are compared regarding the obtained accuracy of trained models. To do so, for each combination of shape and radius a separate data set containing all unique observed neighbourhood patterns is extracted from the replay data set, resulting in a total of 9 data sets. The classification algorithms k-Nearest Neighbour [112], Decision Tree [143], Random Forest [27], AdaBoost [78], and Naïve Bayes [18] are trained for each of the 9 data sets. To ensure a fair comparison a grid-search is used to optimise the parameter settings of each classifier. Stable results are achieved by using 10-fold cross-validation to evaluate the accuracy of each configuration.

The evaluation covers 30 games, 9 data sets, 5 classifiers and a varying number of parameter combinations. Details of the performed grid search can be found in Section A.2. The complete source code and detailed results of this process can be found in the public git-repository [45]. Here, the aggregated results will be shown to discuss the suitability of included algorithms and data sets.

Figure 4.12 shows the accuracy distribution per algorithm and the accuracy distribution per data set on the aggregated results of all games. Results per game are reported in Section A.3. The evaluation shows that the overall best-performing classification algorithm is the Decision Tree. In comparison to both decision-tree based ensemble classifiers, namely the Random Forest and the AdaBoost classifier, the simple decision tree achieves its best results when not being pruned while the ensemble-based classifiers use pruning strategies to avoid overfitting of the training data. However, overfitting can be beneficial in this setting since the training data set is likely

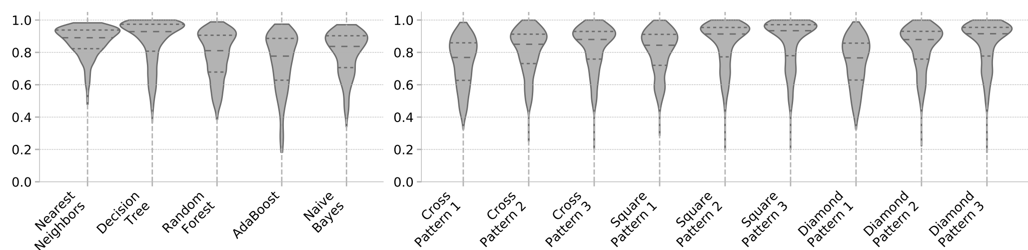


Figure 4.12: Aggregated accuracy values measured on all of the evaluation games: (left) performance distribution per algorithm (right) performance distribution per data set

to cover a large portion of observable patterns. Being able to completely replicate the training data means that the agent would be able to correctly predict every previously observed pattern. Which in return means, that predicting a pattern wrong occurs at most once per pattern.

At the same time, the trained classifiers performed better when using patterns with a larger radius. This is the case because increasing the neighbourhood radius also increases the number of uniquely observable patterns, therefore, providing more training examples to the training process for the same number of observed transitions. Nevertheless, this also increases the number of input variables for the classifier, and therefore, slows down the training process and may result in overfitting. This results in a trade-off, which should be considered when choosing the neighbourhood radius.

Table A.2 further shows the best-performing combination of data set, algorithm, and its parameters per game. The results show that the combination of an unpruned decision tree and the square neighbourhood pattern with a radius of 3 achieves the highest performance in most games. The following evaluation will focus on the agent's game-playing performance when using a pre-trained model. Therefore it is acceptable to achieve an increase in the model quality at the expense of the training time.

4.10.2 Accuracy of Object-based Forward Models

The GVGAI framework provides access to a wide range of variables per object. Each object receives a unique id which allows it to be tracked over the course of the game. Additionally, the object is assigned an object type. Elements of equal object type share the same underlying model, and therefore, exhibit a

similar behaviour. While the object id allows to detect the creation, update, and destruction of objects, the object type will be used to group instances of the same type into separate training data sets.

When applying the model, a two-fold prediction process is applied to each object. First, it will be predicted if the object remains in the next state or will be destroyed. In case of the former, the object's changes in position will be predicted. If the object at hand is the agent's avatar, changes to the avatar's object type are also taken into account. Non-player objects usually do not change the type, but are replaced by a new object of a different type. Since the new object receives a new id, it is not possible to track these changes without an overview of recent collision events, which is not accessible in the GVGAI learning track. Hence, predicting the creation of new objects is currently not considered in this implementation of the object-based forward model.

Both, the parameterisation of classifiers and the choice of input variables of an object-based forward model, have previously been studied in context of the GVGAI framework [56]. Similar to local forward models, unpruned decision trees have proven to perform well in the prediction of upcoming states. Table 4.3 summarises tracked attributes per object. Measuring the model's accuracy by the percentage of objects that were correctly predicted yielded an accuracy of 61.4 – 99.8% depending on the game being played. Hence, the accuracy of object-based forward models seems to be comparable to the accuracy of local forward models.

Table 4.3: Tracked attributes per object instance. Attributes marked with a (*) are only considered for instances of the agent's avatar.

Instance Attributes	Data Type
grid-position	$\mathbb{N}^2$
type of left-/right-/up-/bottom-neighbour	$\mathbb{N}$
player action	{left, right, up, down, use}
distance to avatar	$\mathbb{N}^2$
Target Attributes	**Data Type**
changed position	$\mathbb{N}^2$
has object been destroyed	{True, False}
has object changed its type*	{True, False}

4.11 Evaluation of the Game-Playing Performance

During the following experiments, the agent's game-playing performance will be evaluated. For this purpose, three training setups will be differentiated.

In the first setting, a model will be trained before the evaluation starts. This model will remain constant during the evaluation of the agent. Therefore, the agent will not be able to adapt the model according to new observations. In case of a prediction error, the agent will repeat the error until the end of the evaluation. The detailed evaluation of the constant model will be presented in Subsection 4.11.1.

The second setting consists of an agent that is continuously updating its model of the environment according to newly observed interactions. At the beginning of this evaluation, the agent starts with no model of its environment and needs to build such a model during the evaluation. Over time the agent should be able to adapt to its new environment, improve the accuracy of its learned forward model, and in return, play the game better over time. The evaluation of the continuously training agent will be presented in Subsection 4.11.2.

In the third setting, the agent will be allowed to train a forward model on a subset of levels. However, during the evaluation, the agent will be tested on previously unseen levels of the same game. This setup requires the agent to learn a model which generalises well on unseen examples, and therefore, the agent's ability to transfer its knowledge to new levels. Details about the transfer learning evaluation and its results will be presented in Subsection 4.11.3.

4.11.1 Constant Model

In the first evaluation setting, the game-playing performance of agents which use a pre-trained forward model will be compared. The search algorithms breadth-first search (BFS), rolling horizon evolutionary algorithm (RHEA), and monte carlo tree search (MCTS) will each be combined with either the local forward model (LFM) or the object-based forward model (OBFM) which results in six prediction-based search agents. Additionally, the random

agent is included as a baseline, since it has shown to be one of the best performing agent models in previous years of the GVGAI competition's learning track [133].

First, the respective models are trained for each game by observing the interactions of a random agent. Here, the random agent is playing each level for 2000 ticks each, while the level is restarted after the game has been won or lost, or in case the first 300 interactions have not resulted in either of these two outcomes. Restarting after a fixed amount of ticks increases the diversity of the data set since many games allow the agent to get stuck in unsolvable situations.

After this initial training phase, an agent's game-playing performance will be evaluated by letting it play each of the five levels per game for ten times. Similar to the evaluation of the GVGAI competition's learning track, the agents' performance will be measured based on their average win-rate, their average score, and their average number of ticks until a level has been won or lost. Based on these metrics agents will be ranked according to their average win-rate which should be maximised. In case a total order of the agents cannot be determined this way, a tie-breaker rule is applied. To break such ties, the average score will be used as a second criterion. The third criterion will be the average number of ticks until a level has been won and the final tie-breaker is the average number of ticks until a level has been lost. Agents should maximise their score, minimise the number of ticks until they win, and maximise the number of ticks until they lose. The latter is used to differentiate the agents' game-playing performance in games that can be lost quickly. Avoiding an early loss often indicates that the agent is able to recognise dangerous game-states. If this process does not yield a unique ranking, each of the tied agents will share the same rank.

A summary of achieved ranks per agent is shown in Table 4.4. As in the GVGAI competition, the Formula 1 scoring system is used to aggregate the results. Hence, the best agent receives 25 points, the second-best 18 points and the following ranks 15, 12, 10, 8, and 6 points. This scoring system highly rewards the best player per game, but also requires agents to score well on multiple games to achieve a high score.

The aggregated results show that all agents using a trained forward model overall outperformed the random agent. The best-performing agent

Table 4.4: Aggregated ranks over all tested games and final score per agent

Agents		Rank 1st	2nd	3rd	4th	5th	6th	7th	Formula-1 Score
Random		4	0	1	3	3	4	15	303
LFM	BFS	10	5	4	4	2	2	3	502
	RHEA	3	3	3	8	3	10	0	380
	MCTS	5	3	8	4	2	4	4	423
OBFM	BFS	6	8	2	4	4	1	5	450
	RHEA	3	5	6	2	10	4	0	411
	MCTS	5	7	4	3	5	4	2	441

was the BFS agent using a local forward model, closely followed by the BFS using an object-based forward model. Together they ranked first in 16 out of 30 games. The MCTS-based agents performed slightly worse but often ranked second or third. Of the set of prediction-based agents, the RHEA agent performed worst but was still able to outperform the random agent in many games. Overall, the random agent proved to be the best agent in 3 out of 30 games and was the worst agent in half of the tested games.

Ranking the agents of the same forward model type yields the same ranking order. For both forward models, the BFS agent achieved the highest score, the MCTS agent the second place, and the RHEA agent was third. This can either indicate the general suitability of the algorithms for tested games or be the result of the parameterisation of the individual algorithms.

Reviewing the set of evaluation games indicates that 25 out of 30 games are deterministic[3]. In previous evaluations of the GVGAI competition's single-player planning track, BFS has shown to perform well for deterministic games, e.g. the agent “Yolobot”, winner of the 2014 competition, used BFS for deterministic games and MCTS for non-deterministic games [136]. Since most of the tested games are deterministic, the BFS agent may have an advantage in this evaluation.

The heuristic search methods RHEA and MCTS were configured to use the same number of forward model calls. While the RHEA agent uses a fixed horizon which limits its search depth, the MCTS agent used the same search depth for its rollouts. However, rollouts of MCTS can reach deeper levels of

[3]Non-deterministic games: chase, deceptizelda, fireman, shipwreck, and whackamole

	bait	catapults	chainreaction	chase	chipschallenge	clusters	colourescape	decepticoins	deceptizelda	doorkoban	escape	fireman	garbagecollector	hungrybirds	iceandfire
Random	7	7	7	6	3	7	7	4	4	6	7	5	7	5	7
LFM-BFS	3	3	4	7	6	2	2	1	1	7	3	2	2	1	1
LFM-RHEA	6	6	6	4	2	6	1	2	6	4	1	4	4	6	4
LFM-MCTS	4	4	3	3	7	3	3	3	2	3	6	7	1	4	3
OBFM-BFS	1	1	1	2	4	4	5	7	7	1	5	6	5	2	2
OBFM-RHEA	5	2	5	5	1	5	6	6	3	5	2	1	6	3	5
OBFM-MCTS	2	5	2	1	5	1	4	5	5	2	4	3	3	7	6

Figure 4.13: Agent comparison by rank per game.

the game-tree since they are not limited to start at the root node. The BFS agent was configured to use a smaller number of forward model call but was allowed to prune similar states. This process can result in a higher search depth in case multiple action sequences yield the same result which is often the case in maze-like games.

A more fine-grained analysis can be achieved by analysing the agents' ranking per game which is shown in Figures 4.13 and 4.14. Details on each agent's performance per game can be found in Section A.4. To support the following analysis a hierarchical clustering using complete linkage has been performed. Based on the clustering shown in Figure 4.15 several subgroups of games, in which tested agents ranked similarly, were identified.

In games such as "iceandfire", "labyrinth", and "labyrinthdual", the agent traverses mazes of differing difficulty. The simplest of these games is "labyrinth" in which the agent tries to reach the goal as fast as possible. Apart from the player character, all game components remain unchanged by the agent's actions. In case the player's movement can be predicted correctly by the forward model, the search depth becomes the only limiting factor. As soon as the agent's search range reaches the goal, the agent will walk the shortest possible path. Since the paths in the maze often allow the BFS search to prune states in which no change occurs (walking into a wall), the

	islands	labyrinth	labyrinthdual	painter	realsokoban	run	shipwreck	sokoban	surround	tercio	thecitadel	thesnowman	vortex	watergame	whackamole
Random	7	7	5	4	7	6	7	6	1	1	1	1	7	7	7
LFM-BFS	3	1	1	2	4	5	4	7	4	1	6	5	1	1	1
LFM-RHEA	6	4	3	3	6	4	6	5	3	1	4	6	2	5	5
LFM-MCTS	1	3	2	1	5	7	1	4	5	1	7	2	6	6	6
OBFM-BFS	2	2	4	7	2	1	3	2	7	1	5	7	2	3	4
OBFM-RHEA	4	5	6	5	3	3	5	3	2	1	2	3	5	4	2
OBFM-MCTS	5	6	7	6	1	2	2	1	6	1	3	4	2	2	3

Figure 4.14: Agent comparison by rank per game.

fixed number of expansions resulted in the highest search depth. Both, the RHEA and the MCTS agent randomly walk around the maze until the goal gets in the range of the agent's search depth. From this point on they run directly towards the target. In these games, the random agent is sometimes able to reach the goal by randomly walking around the maze. However, it is less often successful and requires more time than tested prediction-based search agents.

Agents using an LFM have strictly been better than their competitors in the games "colourescape", "decepticoins", "labyrinthdual", and "painter". The games decepticoins and painter includes elements that appear during the course of the game which need to be predicted to be successful in playing this game. Since the implemented OBFM does not predict the creation of new objects, agents using this model were outperformed by agents using a local forward model. Overall, the orange cluster indicates games in which agents using an LFM performed better than agents using an OBFM.

In contrast, agents using an OBFM outperformed LFM-based agents in the games "realsokoban", "run", and "sokoban". In the game "run", the agent needs to run away from a flood of water. Since the OBFM has information on which object represents the agent's avatar it can better prioritise to keep the avatar alive. The other two Sokoban-like games require the agent to push

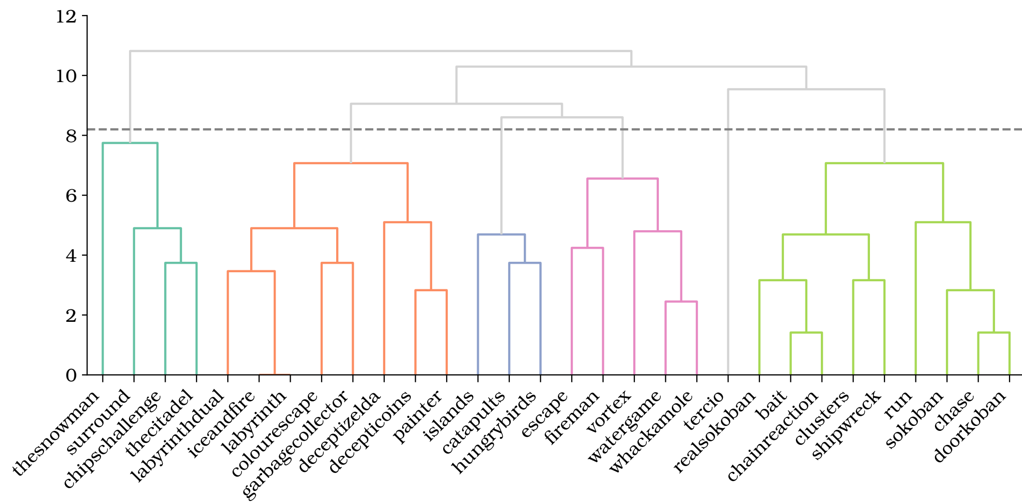


Figure 4.15: Hierarchical agglomerative clustering of each game's ranking vector using complete linkage. The horizontal cut was chosen such that identified clusters can still be interpreted.

blocks on a target tile. Here, previous experiments have shown that a LFM requires a special encoding in case the game is non-Markovian [52]. Since moving blocks obstruct the underlying tiles, the agent cannot predict which tiles will be exposed after moving the block. Changing the encoding of the game-state can help to overcome this problem, but has not been done here for a fair comparison of both forward model types. On average, OBFM agents performed better in games of the green cluster. The games "chase" and "doorkoban" require the agent to be aware of global dependencies between the objects, e.g. the switches in doorkoban change the state of a certain door but there is no local dependency between a switch and a door.

Studying the results of the game "chipschallenge" allows for another interesting comparison. Here, agents using a RHEA-based search were outperforming other agents and the random agent became the third-best agent. In this game, the agent is tasked to collect coins and coloured potions. Collecting a potion adds it as a resource to the agent's inventory and allows the agent to destroy blocks of the same colour. Since the implementations of both forward models did not take the agent's resources into account, it was not possible to predict the destruction of these blocks. Therefore, the agent believes to be trapped. The RHEA agent included enough randomness to destroy coloured blocks by randomly walking into them, but was still

more efficient than the random agent when coins could be collected. On average the light blue cluster includes games in which the agent benefits from random actions. In contrast, the dark blue cluster indicates games in which randomness hinders the agent from playing the level effectively, e.g. the agent loses the game when touching a water tile in the games “islands” and “catapults”.

Overall, all the proposed agents were able to outperform the random agent in most games. Exceptions are the games “surround”, “tercio”, “thecitadel”, and “thesnowman” in which the random agent reached either the first or second rank. In case of the game “tercio”, all agents performed similarly bad by neither winning the game nor scoring any points. While trained forward models were able to predict the next state with high accuracy (cf. Section 4.10), the random agent was never able to score a point during the models’ training phase. Therefore, the trained scoring models were not able to learn to predict which actions yield a reward. Used simulation-based search agents chose actions based on their expected discounted return. Since the reward of every action was always predicted to be 0, all of the actions were equally good resulting in the agent acting randomly. Similar problems were observed in games that only provide a sparse reward and are hard to win for a random agent, e.g. “surround”, “thecitadel”, and “thesnowman”. Due to the sparse reward scheme, the scoring model did not learn to predict future rewards with high accuracy. This resulted in mostly random behaviour for all the tested agents.

The games “chase”, “garbagecollector”, “fireman”, and “whackamole” confront the agent with the trade-off of maximising its reward as fast as possible and keeping itself alive. By maximising the expected discounted return, the multi-objective problem is turned into a single-objective problem. Instead, a multi-objective search could be used to optimise the agent’s actions according to the multi-objective problem and may yield further improvements in game-playing performance. Multi-objective MCTS has shown to outperform single-objective search algorithms in games of the GVGAI framework [134] and multi-objective physical travelling salesman problems [129].

4.11.2 Continuously Trained Model

Since the previous evaluation has shown that tested games can be played with a pre-trained model, the second evaluation will provide further insights into the agents' model learning process. Therefore, a forward model will continuously be trained while the agent is playing the game. To assure that a useful model can be learned, this evaluation will mainly consider games in which the pre-trained model has already was shown to be useful.

In this evaluation, the agent will start without a forward model. While playing each of the 5 levels ten times, the agent can observe the results of its interactions and update its forward model accordingly. Levels will be played in cyclic order to increase the diversity of observed patterns. Playing each level once will be called an epoch. To keep the number of learning processes low, the model will only be updated every 100 ticks and at the end of each level. Until the first forward model has been trained, the agent will choose actions at random.

To measure the change in the agents' game-playing performance, the average win-rate, the average score, and the average number of ticks for winning or losing a level will be measured for each epoch. Detailed results of this measurement can be found in Section A.5. Furthermore, agents are ranked according to their overall performance per level. Figure 4.16 shows the agents' ranking per game in the continuous learning evaluation. An overview of achieved ranks per agent is shown in Table 4.5. Despite training the forward model during the evaluation, the local forward model yielded the best results. The object-based forward model achieved the second-best performance and the random agent came in last.

Comparing the results of the continuous learning process with the pre-trained model setting indicates that most rankings are the same. However,

Table 4.5: Aggregated ranks over all tested games and final score per agent

Agents		Rank 1st	2nd	3rd	Formula-1 Score
Random		1	5	9	250
LFM	BFS	9	4	2	327
OBFM	BFS	5	6	4	293

	bait	catapults	chipschallenge	clusters	decepticoins	escape	garbagecollector	hungrybirds	iceandfire	islands	labyrinth	labyrinthdual	painter	run	watergame
Random	2	3	1	3	2	3	3	3	3	3	2	3	2	3	2
LFM-BFS	1	2	3	2	1	1	2	1	1	2	1	1	1	1	3
OBFM-BFS	3	1	2	1	3	2	1	2	2	1	3	2	3	2	1

Figure 4.16: Ranking of the agents' performance in a continuous learning setup. During the evaluation, the forward model learning agents update their model every 100 ticks or in case a level has ended.

some differences exist which will be shortly discussed to understand why they occur. When continuously updating the forward model, the object-based forward model performed worse than other approaches while playing the games "bait" and "labyrinth". In the game "bait", the agent needs to collect a key and reach the goal, either of these two actions yields a small reward. Until the agent collected any key, the forward model is not able to predict any changes in the score. A similar problem occurred in the game "labyrinth" in which the agent needs to reach the goal to get a point.

The evaluation results indicate that the agent using an object-based forward model did not receive any reward during its first attempts of playing these games, thus, hindering the agent to find any differences in an action's expected score. This indicates the importance of a representative training data set. In case certain events that can occur in the environment are not represented in the training data, the insufficiently trained forward model will stop the agent from performing certain actions since their predicted result seems inferior to other actions. Measuring the confidence in a prediction may allow the agent to select exploring actions for improving the forward model more efficiently.

In the game "chipschallenge" the random agent still performed better than the other two algorithms. Since collected resources cannot be appropriately modelled, the forward model agents will consider doors to remain closed when touching them. Since the BFS agent does not include any randomness in its action selection, the agent will be trapped.

4.11.3 Transfer Learning

In the third evaluation setting, the agent's ability to transfer a model trained on a selection of training levels to a set of evaluation levels will be tested. This evaluation will be done using the same 15 games of the continuous learning setting to assure that a useful model can be learned when being able to train on all five levels.

Here, the agents' training will be restricted to the first three levels in which a random agent is used for a total of 5000 ticks each to generate a training set. To create a diverse set of interactions, the game will be reset after the game has been won or lost, or in case the first 300 interactions have not resulted in either of these two outcomes. After the three training levels have been played, the agent trains a model based on all previously observed interactions. During evaluation the agent will play two previously unseen levels of the same game for ten repetitions each during which the model will remain unchanged.

Both, the local forward model and the object-based forward model, can be used for transfer learning since learned models are independent of the number of observable components. Using a local forward model the same local transition function will be applied to every tile in the game-state. Changing the level's layout or size does not influence the local forward model's applicability. However, in case the evaluation levels include neighbourhood patterns or tiles that were not observed during the training levels, the model's prediction of the next-state may be inaccurate. The object-based forward model predicts the next game-state by predicting changes of every observed game component in the environment. In case a previously unobserved object type is observed, the object's associated sensor values are predicted to stay constant since no model was trained.

Table 4.6: Aggregated ranks over all tested games and final score per agent

Agents		Rank 1st	2nd	3rd	Formula-1 Score
Random		2	5	8	260
LFM	BFS	4	6	5	283
OBFM	BFS	9	4	2	327

	bait	catapults	chipschallenge	clusters	decepticoins	escape	garbagecollector	hungrybirds	iceandfire	islands	labyrinth	labyrinthdual	painter	run	watergame
Random	1	3	1	3	3	3	2	3	3	2	3	2	2	2	3
LFM-BFS	2	1	3	2	2	2	3	1	1	3	2	1	3	3	2
OBFM-BFS	3	2	2	1	1	1	1	2	2	1	1	3	1	1	1

Figure 4.17: Ranking of the agents' performance in a transfer learning setup.

The random agent's performance will be compared to a BFS using either the generated local forward model (LFM BFS) or the object-based forward model (OBFM BFS). Figure 4.17 shows the agents' ranking per game and a summary of achieved ranks per agent is shown in Table 4.6.

Results show that a prediction-based search using either the local forward model or the object-based forward model is able to outperform the random agent. In contrast to previous evaluations, the agent using an object-based agent performed better than an agent using a local forward model.

Once again, both forward model learning agents were beaten by the random agent when playing the games "bait" and "chipschallenge".

In summary, the results of evaluating the agents' game-playing performance indicate that the proposed agent model can be used to handle games that do not provide access to a forward model. The proposed prediction-based search has shown to predict upcoming states with high accuracy, and therefore, allow the agent to play a variety of games effectively. Agents using a pre-trained model have been successful in playing multiple games of the GVGAI framework. Tests using agents which continuously update their model of the environment have shown that the agent is quickly able to improve in terms of game-playing performance in case the environment features a dense reward distribution. Testing the agents' capabilities of transfer learning has shown that the agents' performance is overall better than a random agent. Nevertheless, the agents' game-playing performance seems to be quite limited in case the trained forward model does not generalise well to new levels of the same game, which highlights the importance of a representative training set.

CHAPTER 5

Predictive State Determinisation

Simulation-based search methods have previously been applied to partial information games with mixed success. Methods such as information set MCTS and ensemble UCT handle the uncertainty in the distribution of cards by analysing multiple completions of the partial information game state. To do this, both methods are keeping track of all states that are still possible and perform simulations on those. Nevertheless, the performance has been shown to be dependent on the influence of hidden values in the state observation [191].

A common constraint of these two algorithms is the assumption that new information in the partial observation suffices to render certain states impossible. At the start of the search process, a complete information state is determined by uniformly sampling from the set of states that still comply with observed values. In games where the remaining states are good approximations of the true game state or the number of remaining states is small enough to allow for a thorough exploration of all of them, these algorithms have shown to perform better than standard MCTS and UCT.

This has shown to be the case for the trick-taking card game Doppelkopf[1]. Here, a player's action can reveal additional information on one's remaining hand cards, due to certain restrictions in the action set. Similar to the well-known card game Uno[2] the player can be forced to play cards of the same suit. If this is not possible for a Doppelkopf player, the trick is lost

[1] tournament rules of the "Deutscher Doppelkopf-Verband e.V." http://www.doko-verband.de/Regeln__Ordnungen.html

[2] https://www.letsplayuno.com/

and any card can be played. Hence, in case the agent observes that its opponent is not able to play a certain suit, a large amount of the state space can be rendered impossible. In fact, all card distributions that include the opponent to have any card of the suit can be removed from consideration. Due to this, the agent can be very successful in the end-game, when only a small amount of card distributions are still possible. In comparison to expert players, the agent seems to be missing opportunities for refining its estimation of the remaining state space. However, this does not deviate too much from a uniform sampling such that the general assumption of the applied algorithms seems to be fulfilled to such degree that the performance is not noticeably affected. Hence, they are well-suited for this kind of game.

In the following, a type of game in which the assumptions of information set MCTS and ensemble UCT seem to fail, namely deck-building games, will be discussed. These games consist of many unique cards, of which the players select a subset to build their decks. Two players can use their decks to play against each other allowing for an enormous variety of combinations. Magic: The Gathering[3] and Hearthstone[4], are popular instances of collectible card games and are played by millions of players world-wide[5]. The great appeal of these games is their constantly growing collection of cards[6], which steadily increases the complexity and variety of these games.

Although each deck-building game is unique, further explanations in this thesis will be based on the game Hearthstone. As of 2016 Hearthstone is the leading digital collectable card game with yearly revenue of 395 million US dollars [164]. Due to its digital nature, many data sets are available. The following section will shortly summarise the game's ruleset as well as its challenges to the development of a Hearthstone playing AI.

[3] https://magic.wizards.com

[4] https://playhearthstone.com

[5] according to the Guinness World Record: https://www.guinnessworldrecords.com/world-records/most-played-trading-card-game

[6] e.g. Magic the Gathering has about 24000 cards and Hearthstone about 2000 cards (as of 1st of October 2019)

Figure 5.1: Elements of the Hearthstone game board (bottom: player, top: opponent): (1) weapon (2) hero (3) opponent's minions, (4) player's minions, (5) hero power, (6) hand cards, (7) mana, (8) decks, (9) history of recent events

5.1 Hearthstone: Heroes of Warcraft

Hearthstone is a turn-based digital collectible card game developed and published by Blizzard Entertainment. Players compete in one versus one duels using self-constructed decks, each belonging to one hero out of nine available hero-classes. In those matches, players try to beat their opponents by reducing their starting health from 30 to 0. This can be achieved by playing cards from the hand onto the game board at the cost of *mana*. Played cards can be used to inflict damage to the opponent's hero or to destroy cards on his side of the game board. At the start of the game, the player who goes first draws three cards and the player who goes second draws four cards. To further balance the game the second player is given a special card called "The Coin", which increases the mana by one for a single turn. The amount of mana available to the player increases every turn (up to a maximum of 10). More mana gives access to increasingly powerful cards and increases the complexity of turns while the game progresses. At the beginning of each turn, the player draws a new card except for the case that their deck is empty, in which case they receive a step-wise increasing amount of *fatigue*-damage. The standard game board is shown in Figure 5.1.

Players need to construct decks of 30 cards, which can be chosen out of the 2150 currently obtainable cards [21]. Cards can be included once or

a) Minion card b) Spell card c) Weapon card

Figure 5.2: Examples of general card types. Cards include (1) mana cost, (2) attack damage, (3) health/durability, (4) special effects, (5) and minion type.

twice depending on their rarity. However, these cards need to be unlocked by aquiring card packs, which can either be earned while playing the game or bought in the ingame store. New cards are frequently added in game updates. The time between two of these updates is referred to as a patch period. Each card bears unique effects, which the players can use to their advantage. Additionally, each player chooses a hero, which gives access to a class-specific pool of cards and its associated hero power. This power can be played once per turn for the cost of 2 mana.

Cards can be of the type minion, spell, or weapon. Figure 5.2 shows one example of each card type. Minion cards assist and fight on behalf of the player. They usually have an attack, health, and mana cost-value, as well as a short ability description. Furthermore, minions can belong to a special minion type, which is the basis for many synergy effects. One turn after they have been played, they can attack the enemy's side of the board to inflict damage on either the opponent's minions or hero. Attacking a target also reduces the attacker's health by the target's attack value. In case any minion's health drops to zero or below, it is removed from the board and put into the player's graveyard. Spell cards can be cast at the cost of mana to activate various abilities and are discarded after use. They can have a wide range of effects, e.g. boosting the attack of your minions.

Secrets, which are a special kind of spells, can be played without immediately activating their effect. After a trigger condition was fulfilled, the secret will be activated. Once activated, the secret is removed from the board. Weapon cards are directly equipped to the player's hero and enable him to attack. Their durability value limits the number of attacks until the weapon breaks. Only one weapon can be equipped at the same time.

Hearthstone decks are often created around a common theme. Multiple cards that positively influence each other can create strong synergies and increase the value of each card in the deck's context. For this reason, the value of a single card highly depends on the player's hand cards, current elements on the board, and the deck in general. Common examples are minion cards of the same type, e.g. "Murloc", which give each other additional advantages, e.g. "Your other Murlocs have +1 Attack". Each of these minions is comparatively weak, but their value increases when they are played together. Generated decks can be categorised into three major categories: aggro, control, and mid-range. Aggro decks build on purely offensive strategies, which often include a lot of minions. Control decks try to win in the long run by preventing the opponent's strategy and dominating the game situation. The playing style of mid-range decks is between aggro and control. They try to counter early attacks to dominate the game board with high-cost minions in the middle of the game. Game length and branching factor can be dependent on the player's decks in the current game. Some decks try to play single high-cost cards, whereas others build on versatile combinations. The complexity of each turn and the uncertainty faced during the game makes Hearthstone a challenging problem for AI research.

Hearthstone's AI was designed by a small team of developers and focusses on a fun and engaging experience [153]. In its current state, it is unable to compete with expert players. A recent paper discussed the many challenges for developing a Hearthstone AI [82]. Key problems mentioned in this paper include the hidden information on the opponent's hand and deck cards, the stochasticity of the initial shuffling and the card draw, as well as the randomness of certain card effects. Especially the prediction of the opponent's cards will be a necessary requirement to increase the search depth of AI agents. These are currently limited to optimising the agent's turn, but cannot effectively go beyond that. Predicting the opponent's deck

is a challenge in itself since the deck building process in Hearthstone allows players to choose nearly any set of cards as long as they can be played by the chosen hero. Observing a card to be played does, therefore, not allow to infer anything on the other cards' possibility. In this case, the pruning of impossible states by information set MCTS's remains ineffective. Nevertheless, expert players have shown to reach win-rates above average, partly due to their ability to guess what their opponent is planning. Therefore, detectable and exploitable patterns seem to exist.

In the following, it will be explained how exploitable patterns can emerge. In Hearthstone two-game phases can be distinguished. The classic game-playing phase consists of two players playing a match against each other. At this stage, both players already selected their deck's, which remain unknown to their upcoming opponent. While the average win-rate of a deck against another deck can be estimated by taking a history of previous games into account, the outcome of this match will be determined to a large degree by the players' skill and the luck of the draw. Therefore, the game-playing phase is about how two players play. Nevertheless, another game phase, the meta-game phase, is "played" before the actual game-playing phase. The meta-game describes a game of deck selection against the entirety of possible opponents. Due to the fact, that the average win-rates of any two decks playing against each other diverges from 50%, some decks could exist that exhibit a win-rate above average. Thus, selecting these decks for play becomes more attractive to players. This results in an evolutionary dynamic in which certain decks are picked more often than others. Therefore, the meta-game is about what is being played. While being separated in time, both phases strongly influence each other.

In a work by Burszstein [30] the meta-game is considered to provide us with the hints about likely game states, due to the following reasons:

- **Card power:** Some cards are inferior versions of other cards. Therefore, they are less interesting to be included in a player's deck.

- **Card synergies:** It is more likely to observe decks that exploit card synergies since they will be more successful than decks that do not exploit them. Rational players will tend to play successful decks.

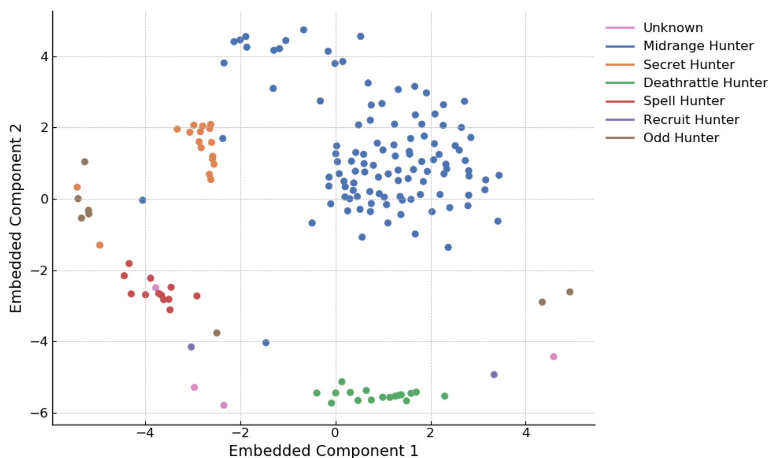


Figure 5.3: 2D-embedding obtained by Multi Dimensional Scaling of the deck space of the Hunter hero class. Points of similar colour belong to decks of the same archetype.

- **Deck archetypes:** The meta-game quickly converges to a small set of deck archetypes. A deck archetype is an abstract representation of a set of similar decks. These similarities occur since successful decks are often copied by other players. Copied decks can include minor variations since players may not possess all the necessary cards.

Especially relevant seems to be the emergence of deck archetypes. These popular deck prototypes are often describing a general strategy. They include essential cards, called core cards, and variant-cards that may be replaced for adapting to the player's style. The success of these deck archetypes results in a highly clustered deck space. Figure 5.3 shows a 2D-embedding of the deck space of the Hunter hero class. Only a few outliers can be detected, which may be due to errors in the human labeling process.

In the following, two methods for predicting the opponent's cards based on previously observed cards will be proposed. The first method is a heuristic based on the concept of card synergies and exploits their frequent co-occurrence in decks and game-traces (see Section 5.2). The second method adds a layer of abstraction to the card prediction process by not modelling upcoming cards, but predicting the current deck archetype in play (see Section 5.3). The agent model utilising either of these two prediction models will be presented in Section 5.4 after which the evaluation is described in Section 5.5.

5.2 Card Sequence Models

Predicting upcoming cards can be done by correctly estimating the probability of each card to be observed in the upcoming turns. Without further knowledge, it needs to be assumed that every possible card (see deck-building restrictions) can appear with the same probability during the next turn. Such a uniform distribution is far from the reality since characteristics like the popularity of cards or deck archetypes and all previously observed actions can influence this probability distribution. In this section, it will be discussed why the problem is too complex to be solved by estimating the true probability distribution of upcoming cards and how a heuristic solution can overcome this problem.

The complexity of Hearthstone's probability distribution is enormous. This should be exemplified using a simplified probabilistic model, which is based on the sequential nature of card-games but ignores the influence factors of the opponent or the meta-game. For this purpose, it is assumed that the probability of observing a card c_t at time t is dependent on the sequence of previously observed cards $(c_1, c_2, \ldots, c_{t-1})$. Hence, it is necessary to estimate and store the parameters of the conditional probability distribution denoted by $P(c_t \mid c_{t-1}, \ldots, c_1)$. The dependent structure of this distribution can be justified by card synergies, which influence the strength of other cards.

The number of parameters to store this distribution only for the last card to be played (ignoring card effects that allow playing additional cards) is about $2000^{30} \approx 1.07 \cdot 10^{99}$. Including previously ignored influence factors would make this problem even more complex, therefore, further increasing the number of parameters. Since the number of parameters in this simplified model already exceeds the number of atoms in the universe[7] storing or processing this probability distribution becomes infeasible. Hence, a heuristic approach is chosen to approximate the likelihood of upcoming cards.

For this purpose, it will be assumed that each card of the sequence of previously observed cards provides independent information on the next card to be played. The general heuristic to be implemented is that cards that appear frequently together will do the same in the future. The idea of using

[7] estimated number of atoms in the universe $\approx 10^{86}$ [178]

this card co-occurrence scheme is motivated by a work by Elie Bursztein [30]. Here, replay data is analysed to count the co-occurrence of cards in play traces of a single player. Information on the other player does not influence the counting and the succeeding prediction.

In this work, five different counting schemes will be analysed regarding their prediction accuracy of upcoming cards. Each of the counting schemes will be shortly introduced:

Isolated Turn (isolated) The isolated turn bigram counting scheme (further referred to as *isolated*) will increase the co-occurrence counter for all card pairs that were played during the same turn. This will be relevant to detect and predict card combinations that are frequently played together in the same turn. One example is the coin-card in Hearthstone, which adds 1 mana to the player's mana pool. Since it will be known that the card remains on the opponent's hand, the isolated bigram counting may be especially relevant to predict these scenarios.

Successive Turn (successive) As the name of the successive turn bigram counting suggests, card combinations that occur in successive turns will be counted. In contrast to the isolated bigram counting, the successive bigram counting is not symmetric. The counter for (a, b) will be increased by one if and only if card a was played one turn before card b. This may be especially accurate during the first turns in which players try to play *on curve*, meaning in each turn they try to play cards that cost the maximal amount of mana they could currently spend.

Combined This bigram counting scheme combines the two previous schemes by adding their results. Since both previous methods are quite limited in the number of co-occurrences counted per game it may be beneficial to combine their results to not just represent both concepts in a single bigram database, but also be more efficient in the number of games to be analysed.

Game Sequence To further boost the number of analysed bigrams per game, the game sequence bigram counting scheme will increase the counter of all pairs (a, b) for which the card a was played before card b.

Whole Game Finally, the whole game counting scheme adds one to all pairs of cards that were played in the same game by the same player independent of their order of play. This way a large number of co-occurrences can be stored per game such that the replay data can be used more effectively. The whole game bigram counting scheme may be able to detect deck archetypes by checking for cards that frequently co-occurred with all previously observed cards. Therefore, it may be more effective during the late-game. This bigram counting scheme is similar to the one used by Elie Burszstein [30].

Based on the way the bigram counting schemes were defined the following principle holds for all pairs of cards $c_i, c_j \in X$:

$$\begin{aligned} f_{isolated}(c_i, c_j) \; + \; f_{succeeding}(c_i, c_j) &= f_{combined}(c_i, c_j) \\ f_{combined}(c_i, c_j) \leq f_{game_sequence}(c_i, c_j) &\leq f_{whole_game}(c_i, c_j) \end{aligned} \tag{5.1}$$

whereas $f_{method}(c_i, c_j) \in \mathbb{N}$ describes the number of co-occurences of the chosen bigram counting scheme. Note, that depending on the used bigram counting scheme the value of $f(c_i, c_j)$ does not need to be equal to $f(c_j, c_i)$. Only the isolated as well as the whole game bigram counting schemes satisfy symmetry of co-occurrence values.

The following example will be used to better differentiate the results of each bigram counting scheme. Consider a game of three turns in which the cards a, b, c, and d were played by the same player in the following order:

$$\underbrace{a}_{\text{turn 1}} \Rightarrow \underbrace{b, c}_{\text{turn 2}} \Rightarrow \underbrace{d}_{\text{turn 3}} \tag{5.2}$$

The proposed counting schemes result in the following bigrams:

Table 5.1: Exemplary results of proposed counting schemes.

counting scheme	resulting bigrams			
isolated	a:{}	b:{c:1}	c:{b:1}	d:{}
successive	a:{b:1, c:1}	b:{d:1}	c:{d:1}	d:{}
combined	a:{b:1, c:1}	b:{c:1, d:1}	c:{b:1, d:1}	d:{}
game sequence	a:{b:1, c:1, d:1}	b:{c:1, d:1}	c:{b:1, d:1}	d:{}
whole game	a:{b:1, c:1, d:1}	b:{a:1, c:1, d:1}	c:{a:1, b:1, d:1}	d:{a:1, b:1, c:1}

5.2.1 Prediction of Upcoming Cards

The prediction of upcoming cards will be done by combining the bigram co-occurrence values of the given card and each of the previously observed cards. Hence, the total co-occurrence value $f(c)$ of a card $c \in X$ given a sequence of cards $(c_1, c_2, \ldots, c_t)$ can be determined by

$$f(c) = \sum_{i=1}^{t} f(c_i, c) \tag{5.3}$$

All cards can be ranked by their total co-occurrence value. Thereafter, the top-k ranked cards can be predicted to be the most likely cards to be observed.

Alternatively, let the probability $P(c)$ of a card to appear in the upcoming turns be given by

$$P(c) = \frac{f(c)}{\sum_{c' \in X} f(c')} \tag{5.4}$$

The resulting probability distribution can be used to sample upcoming cards. Nevertheless, this would allow a card to be included multiple times. Since players are only allowed to include each card twice in a deck, the second method could yield predicted game-states that are impossible. This problem can be resolved by either repeating the sampling process in case a card was sampled more than 2 times or excluding each card that was sampled twice and recalculating the probability distribution.

Since the proposed method includes drastic simplifications on the complexity of the underlying probability distribution, predicted cards are like to diverge from the true probability distribution. This estimate could be improved by the introduction of tri- or n-grams. Nevertheless, due to the increasing number of possible tri- and n-grams estimating their co-occurrence counts will need a larger amount of data and storing them a much higher amount of memory. An evaluation by Elie Burszstein [30] has shown that using tri-grams instead of bigrams resulted in reduced prediction accuracy when using the same amount of training samples.

5.3 Clustering-based Meta-game Analysis

As discussed previously, modeling the probability of an upcoming card, based on all previously observed cards is infeasible, due to the possible combination of any card. The resulting probability distribution is too fine-grained to be stored, learned, and processed efficiently. Adding a layer of abstraction could solve this problem. This abstraction will be based on the idea of a clustered deck-space, in which frequently observed decks are similar to each other. Due to the convergence of the meta-game, observed cards can be used to predict the deck or deck archetype in play, instead of predicting upcoming cards. Knowing the deck archetype may allow us to predict the opponent's hand cards with reasonable accuracy.

Since the set of viable decks changes with every update of the game, a reliable method for extracting deck clusters from game-play data needs to be found. Closely following my work in [55], this process will be based on fuzzy multisets, which are introduced in the following Subsection 5.3.1. These fuzzy multisets will be used to represent decks, deck clusters, and their deck representatives. Necessary components of the clustering process, such as applied distance functions and exemplary clustering algorithms, will be described in Subsection 5.3.2. Subsequently, a method for generating a cluster prototype is proposed in Subsection 5.3.3. This section ends with a description of the prediction process for upcoming cards based on the extracted cluster prototypes.

5.3.1 Representing Deck Archetypes using Fuzzy Multisets

A deck is a collection of multiple cards, of which each of these cards can be represented multiple times. Multisets (also called bags) can be used as mathematical representations of decks. In the following, the notation introduced by Miyamoto [113] and Yager [186] will be used.

Let a multiset M be denoted by

$$M = \{C_M(x)/x \ : \ x \in X\} \tag{5.5}$$

in which X is the set of elements x that can be included and C_M a function that maps each object x_i to its number of copies n_i in M:

$$C_M : X \to \mathbb{N} \qquad C_M(x_i) = n_i \tag{5.6}$$

For comparing two multisets L and M, inclusion is defined by

$$L \subseteq M \quad \text{iff} \quad C_L(x) \leq C_M(x) \text{ holds } \forall x \in X \tag{5.7}$$

and (as a consequence) equality is given by

$$L = M \quad \text{iff} \quad C_L(x) = C_M(x) \text{ holds } \forall x \in X \tag{5.8}$$

Union, intersection, and addition are defined pointwise for all $x \in X$ by

$$C_{L\cup M}(x) = \max C_L(x), C_M(x) \tag{5.9}$$

$$C_{L\cap M}(x) = \min C_L(x), C_M(x) \tag{5.10}$$

$$C_{L\oplus M}(x) = C_L(x) + C_M(x) \tag{5.11}$$

While a specific deck is always a crisp multiset, this concept seems to fail in case of representing a deck's archetype. Here, it would be desirable to be able to differentiate between the essential core cards and cards that are included due to the player's preferences. Since a deck archetype is a mixture of multiple decks a fuzzy multiset seems to be a good representation for it.

A fuzzy extension of multisets was first introduced by Yager (using the term fuzzy bags) [186]. Here, the sample fuzzy multiset

$$A = \{(x, 0.5),\ (x, 0.3),\ (y, 1),\ (y, 0.5),\ (y, 0.2)\}$$

denotes the occurrence of each object and its membership degree. Specifically, the object x is included twice, once with a membership degree of 0.5 and a second time with a degree of 0.3. For simplicity, objects of the same kind and their membership degrees will be grouped, as demonstrated in:

$$A = \{(0.5, 0.3)/x,\ (1, 0.5, 0.2)/y\}$$

in which the memberships $\{0.5, 0.3\}$ and $\{1, 0.5, 0.2\}$ correspond to the objects x and y, respectively. Therefore, in fuzzy multisets $C_A(x)$ is a finite multiset of the unit interval [186].

For each object x the membership sequence will be defined to be a decreasingly-ordered sequence of elements in $C_A(x)$. The standard form introduced by Miyamoto [113] will be used:

$$(\mu_A^1(x), \ldots, \mu_A^p(x)), \qquad \mu_A^1(x) \geq \cdots \geq \mu_A^p(x) \tag{5.12}$$

Let $L(x; A)$ be the length of the membership sequence $(\mu_A^1(x), \ldots, \mu_A^p(x))$ of multiset A denoted by

$$L(x; A) = \begin{cases} \max\{j \ : \ \mu_A^j(x) \neq 0\} & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases} \tag{5.13}$$

Any operation between two multisets A and B requires the membership sequences of each object to be of equal length. When comparing two fuzzy multisets of different lengths, the remaining membership degree's will be assumed to be zero. For the sake of simplicity, let

$$\mu_A^i(x) = 0; \quad \forall i \text{ with } L(x; A) < i \leq L(x; A, B) \tag{5.14}$$

in case the object x is included less than $L(x; A, B)$ times in the multiset A (likewise for B). Let the length $L(x; A, B)$ of the resulting membership sequence be defined by

$$L(x; A, B) = \max\{L(x; A), \ L(x; B)\} \tag{5.15}$$

Similar to crisp multisets inclusion, equality, union, and intersection can be defined based on the membership sequences of each element. Let A and B be two fuzzy multisets.

$$A \subseteq B \quad \text{iff} \quad \mu_A^j(x) \leq \mu_B^j(x) \text{ holds for } \quad j = 1, 2, \ldots, \ L(x; A, B), \ \forall x \in X \tag{5.16}$$

$$A = B \quad \text{iff} \quad \mu_A^j(x) = \mu_B^j(x) \text{ holds for } \quad j = 1, 2, \ldots, \ L(x; A, B), \ \forall x \in X \tag{5.17}$$

Similarly, union and intersection are defined pointwise for all $x \in X$ by

$$\mu^j_{A \cup B} = \mu^j_A(x) \vee \mu^j_B(x) \quad j = 1, 2, \ldots, L(x; A, B) \tag{5.18}$$

$$\mu^j_{A \cap B} = \mu^j_A(x) \wedge \mu^j_B(x) \quad j = 1, 2, \ldots, L(x; A, B) \tag{5.19}$$

The following short example should be reviewed to clarify the notation. Consider the two fuzzy multisets A and B over the set of objects $\{x, y, z\}$:

$$A = \{(0.5,\ 0.2)/x,\ (1.0,\ 0.5,\ 0.2)/y\}$$
$$B = \{(1.0)/x,\ (0.7,\ 0.6)/y,\ (0.9, 0.5)/z\}$$

The length per object is:

$$L(x; A, B) = 2; \quad L(y; A, B) = 3; \quad L(z; A, B) = 2$$

For simplicity the membership sequences for both multisets will be extended according to the maximal observed length:

$$A = \{(0.5,\ 0.2)/x,\ (1.0,\ 0.5,\ 0.2)/y,\ (0.0, 0.0)/z\}$$
$$B = \{(1.0,\ 0.0)/x,\ (0.7,\ 0.6,\ 0.0)/y,\ (0.9, 0.5)/z\}$$

Union and intersection of both multisets can be determined based on the extended membership sequences:

$$A \cup B = \{(1.0,\ 0.2)/x,\ (1.0,\ 0.6,\ 0.2)/y,\ (0.9, 0.5)/z\}$$
$$A \cap B = \{(0.5)/x,\ (0.7,\ 0.5)/y\}$$

5.3.2 Clustering of Decks

To create an abstract representation of the meta-game, a cluster analysis to extract clusters of similar decks will be performed. As discussed in the previous section each deck will be represented as a multiset of cards. In turn, a mixture of decks or a cluster can be represented as a fuzzy multiset of cards. Each cluster should contain decks that are similar to each other but different from decks of other clusters. For doing so, distance functions need to be defined for measuring the differences between two decks.

Distance Functions

To measure the distance of two multisets L and M their Euclidean distance will be defined by

$$d_{euclid}(L, M) = \left(\sum_{x \in X} (C_L(x) - C_M(x))^2 \right)^{\frac{1}{2}} \tag{5.20}$$

and transfer the definition to be applied to fuzzy multisets A and B:

$$d_{euclid}(A, B) = \left(\sum_{x \in X} \sum_{i=1}^{L(x;A,B)} (\mu_A^i(x) - \mu_B^i(x))^2 \right)^{\frac{1}{2}} \tag{5.21}$$

In this work,results based on the Euclidean distance will be compared to results obtained from applying the Jaccard distance measure. The Jaccard distance can be used to measure the difference between two sets [95].

$$d_{jaccard}(x, y) = 1 - \frac{|x \cap y|}{|x \cup y|} = 1 - \frac{\sum_i \min(x_i, y_i)}{\sum_i \max(x_i, y_i)} \tag{5.22}$$

Here, the definition is extended to also apply to two multisets L and M:

$$d_{jaccard}(L, M) = 1 - \frac{\sum_{x \in X} \min(C_L(x), C_M(x))}{\sum_{x \in X} \max(C_L(x), C_M(x))} \tag{5.23}$$

Similar to the Euclidean distance the equation to measure the distance of two fuzzy multisets A and B will be transferred:

$$d_{jaccard}(A, B) = 1 - \frac{\sum_{x \in X} \sum_{j=1}^{L(x;A,B)} \mu_{A \cap B}^j}{\sum_{x \in X} \sum_{j=1}^{L(x;A,B)} \mu_{A \cup B}^j} \tag{5.24}$$

The distance functions proposed in this work fulfil all the requirements of a metric. Nevertheless, it is unclear if the humanly perceived difference of two decks does the same. Non-negativity and similarity may be easy to satisfy. However, human judgment could fail the concept of identity in case two alternative cards would have equal effects in the context of the remaining cards of a deck. Such a context-driven weighting could result in two deck variants being perceived as more similar to each other than two decks with the same number of card changes. Additionally, the required judging process is especially hard due to the high dimensionality of the deck space. Therefore,

it is in question if the application of a metric distance function is a strong requirement. The application of Pseudo- and Semimetrics may result in clusters that better match human perception. However, designing distance functions for this specific context may require more knowledge of the human judgment process, which is currently not available.

Clustering Algorithms

The process of clustering a set of input patterns $X = \{x_1, \ldots, x_j, \ldots, x_N\}$, in which each pattern $x_j = (x_{j1}, x_{j2}, \ldots, x_{jd}) \in R^d$ is defined by its features x_{ji} (also referred to as dimensions), results in their partitioning into a clustering $C = \{C_1, \ldots, C_K\}$ with $K \leq N$, such that elements of the same cluster are similar to each other and different from elements of other clusters. In case of a hard partitional clustering, the resulting clusters C_1 to C_K should be non-empty, pairwise disjunct, and their union should be equal to the original dataset X [185]. Handl et al. [77] identified three fundamental properties of desired cluster outcomes, namely compactness, connectedness, and spatial separation, that often guide the design of clustering algorithms. In the following, three clustering algorithms that each represent a different concept of clustering will be introduced.

k-means: The clustering algorithm k-means [100] is the most common representative of partitional clustering algorithms. During initialisation, k cluster prototypes are randomly generated or selected from the set of available data points. The cluster prototypes are iteratively updated to better represent and partition the points in the data set. For this purpose, each data point is assigned to its closest prototype. In a second step, the prototypes are moved to the center of all assigned points to minimise the sum of squared errors between a prototype and all its assigned data points.

$$SSE(C) = \sum_{k=1}^{K} \sum_{x_i \in C_k} |x_i c_k|^2, \tag{5.25}$$

such that c_k represents the centroid of cluster C_k. Due to its scoring function, k-means favors clusters that are compact and well separated.

K-means is known to quickly converge to a local optimum. However, its result very much depends on the initial placement of cluster prototypes.

Since the global optimisation of its scoring function is known to be NP-hard [102], k-means is often repeated with different initialisations and the result with minimal SSE is kept.

Hierarchical Agglomerative Clustering: In contrast to the flat result of a partitioning algorithm like k-Means, hierarchical clustering algorithms produce a hierarchy of clusters. Agglomerative (bottom-up) clustering algorithms start by assigning each point to an individual cluster. Consecutively, the two closest clusters are merged until all objects belong to a single cluster. Alternatively, the clustering process can be stopped in case the Jaccard distance of all cluster pairs is 1.0 during a single merge step. In the latter, none of the current clusters have any object in common. Analysing the resulting hierarchy can provide us with hints on the number of clusters.

The linkage criterion describes how the distance between two clusters is measured. Two popular linkage criteria are single and complete linkage. Single linkage [108] uses the minimum pairwise distance in between elements of the two clusters and can be used to find connected structures. In contrast, complete linkage [94] optimises for compact structures by merging the two clusters with the smallest maximum pairwise distance in between their contained elements.

$$\text{single linkage} \qquad d_{single}(C_i, C_j) = \min_{a \in C_i, b \in C_j} d(a, b) \tag{5.26}$$

$$\text{complete linkage} \qquad d_{complete}(C_i, C_j) = \max_{a \in C_i, b \in C_j} d(a, b) \tag{5.27}$$

Density-based Clustering: Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a clustering algorithm proposed by Ester et al. [58]. Here, a cluster represents a dense region in space. The ε-neighbourhood of a point consists of all points with a maximal distance of ε:

$$N_\varepsilon(p) = \{q \in X \mid d(p, q) \leq \varepsilon\} \tag{5.28}$$

The region around a point is considered to be dense in case the number of points in its ε–neighbourhood exceeds the threshold m_{Pts}. Points that satisfy this condition are called core points. A point q is directly density-reachable

from point p, if $q \in N_\varepsilon(p)$ and p is a core-point. The transitive closure of directly density-reachable points is called density-reachable. Finally, two points are density-connected when a point exists from which both are density-reachable. A cluster is described by the maximal set of points that are density-connected to each other.

5.3.3 Modelling Deck Archetypes

In the following, a deck archetype is going to be modelled as a prototypical representation of a deck cluster. Such a deck archetype consists of core and variant cards. In the following, it will be shown that these two card types cannot be distinguished in crisp multisets, but when using fuzzy multisets.

Let us consider two crisp decks D_1 and D_2 over the set of elements $X = \{a, c, d, e, f\}$ of the form:

$$D_1 = \{1/a,,\ 2/b,\ 1/c,\ 0/d,\ 2/e\}$$
$$D_2 = \{1/a,,\ 2/b,\ 0/c,\ 2/d,\ 1/e\}$$

The intersection $M_{D_1 \cap D_2}$ of these two decks is the multiset:

$$M_{D_1 \cap D_2} = \{1/a,\ 2/b,\ 0/c,\ 0/d,\ 1/e\}$$

The resulting set describes the core of these two decks. The information on possible variants is lost when using the intersection of these decks. A similar problem occurs if when generating the union $M_{D_1 \cup D_2}$ of both decks:

$$M_{D_1 \cup D_2} = \{1/a,\ 2/b,\ 1/c,\ 2/d,\ 2/e\}$$

The result of the union operator preserves information on the inclusion of d and e. However, this misleadingly represents the occurrence counts of these variants, i.e. based on its count in $M_{D_1 \cup D_2}$ variant object c is indistinguishable from the core object a (similar observations can be made for the objects b and d). Hence, objects with different inclusion patterns in D_1 and D_2 are equally represented in the merged multiset.

For the crisp multiset the average multiset $M_{\langle L,M\rangle}$ of two multisets L and M will be defined to include the average number of occurrences per object in these multisets and denote it by

$$C_{\langle L,M\rangle}(x) = \frac{C_L(x) + C_M(x)}{2}, \quad \forall x \in X \tag{5.29}$$

Hence, the average of clusters D_1 and D_2 is:

$$M_{\langle D_1,D_2\rangle} = \{1/a,\ 2/b,\ 0.5/c,\ 1/d,\ 1.5/e\}$$

Applying the average operator allows us to clearly distinguish the inclusion patterns for a and c. However, similar results for objects with varying numbers of inclusion, e.g. a and d can still be observed. Extending the representation to fuzzy multisets can help to solve this problem.

For this purpose, average operator for crisp multisets will be transferred to fuzzy multisets by calculating the average of every element of an object's membership sequence. Thus, the average operator for two fuzzy multisets A and B can be denoted by

$$\mu^i_{\langle A,B\rangle}(x) = \frac{\mu^i_A(x) + \mu^i_B(x)}{2}, \quad i = 1, \ldots, p,\ \forall x \in X \tag{5.30}$$

Representing both decks as fuzzy multisets results in the following centroid:

$$\begin{aligned} D_1 &= \{(1)/a,\ (1,1)/b,\ (1)/c,\ (0)/d,\ (1,1)/e\} \\ D_2 &= \{(1)/a,\ (1,1)/b,\ (0)/c,\ (1,1)/d,\ (1)/e\} \\ M_{\langle D_1,D_2\rangle} &= \{(1)/a,\ (1,1)/b,\ (0.5)/c, (0.5,0.5)/d,\ (1.0,0.5)/e\} \end{aligned}$$

Since the result of merging multiple multisets should be independent of their merging order, the definition of the (fuzzy) multiset centroid will be adjusted to satisfy associativity. Specifically, the following properties to be fulfilled should be fulfilled:

$$\begin{aligned} C_{\langle\langle D_1,D_2\rangle,D_3\rangle}(x) &= C_{\langle D_1,\langle D_2,D_3\rangle\rangle}(x), \quad \forall x \in X \\ M_{\langle\langle D_1,D_2\rangle,D_3\rangle} &= M_{\langle D_1,\langle D_2,D_3\rangle\rangle} \end{aligned} \tag{5.31}$$

Let a cluster C be a multiset over the set $\{M_1, \dots, M_n\}$ of multisets over the set of objects X. The centroid $\langle c \rangle$ of cluster C, which itself is a multiset over the set of objects X, should be independent of the order of inclusion of said multisets, thus being an associative operation. First, the pairwise average of two multisets will be replaced with the arithmetic mean over all included multisets. Additionally, the number of inclusions per multiset will be taken into account:

$$C_{\langle c \rangle}(x) = \frac{\sum_{M_i \in C} \left(C_{M_i}(x) \cdot C_C(M_i)\right)}{\sum_j C_C(M_j)}, \quad \forall x \in X \tag{5.32}$$

The same can be done for a cluster of fuzzy multisets:

$$\mu^k_{\langle c \rangle}(x) = \frac{\sum_{M_i \in C} (\mu^k_{M_i}(x) \cdot C_C(M_i))}{\sum_j C_C(M_j)}, \quad k = 1, \dots, p, \quad \forall x \in X \tag{5.33}$$

The cluster centroid will be used to represent the cluster and all its contained decks in a single fuzzy multiset. This way the deck archetype preserves information on core and variant cards as well as taking the times each deck has been played into account. Therefore, it serves as a layer of abstraction of the current meta-game.

5.3.4 Predicting Upcoming Cards

In the previous section, it was explained how a data set of recently played decks can be reduced to a small number of clusters and their representatives. The following prediction method will search for the representative that is most similar to all previously observed cards and use it to predict further cards. Knowing the cluster centroids, a prediction of the opponent's cards can be made using the following multi-step process:

1. **Construct a fuzzy multiset of observed cards:** At the beginning of the game the agent starts with an empty fuzzy multiset. During the opponent's turn, the agent keeps track of all the opponent's actions. Each card played is added to the Fuzzy Multiset with a membership grade of 1.0. In case the card has previously been played the membership sequence of this card is extended by another entry of 1.0.

2. **Determine the most likely deck archetype:** During the agent's turn, the agent first needs to determine the most-likely deck archetype. This can be done by calculating the pair-wise distance between the fuzzy multiset of observed cards and all deck archetype representatives that are themselves fuzzy multisets. The closest centroid is assumed to be the most likely deck archetype and further considered for the card prediction. It is also possible to select a deck archetype based on each centroids distance value. For this purpose, all distances $d(c_i, obs)$ between the observation obs and the fuzzy centroids $\langle c \rangle$ are first transformed into a similarity value by

$$sim(\langle c_i \rangle, obs) = 1 - \frac{d(c_i, obs)}{\max_i(d(c_i, obs))} \tag{5.34}$$

The resulting similarity values are further transformed into a probability distribution by

$$P(\langle c_i \rangle) = \frac{sim(\langle c_i \rangle)}{\sum_{j=1}^{K} sim(\langle c_j \rangle)} \tag{5.35}$$

Instead of choosing the closest centroid, the resulting probability distribution can be used to sample a deck archetype. Only the most probable deck archetype will be considered in the upcoming evaluation of the prediction accuracy. However, during the agent's search it may be advantageous to extract cards from a variety of possible archetypes.

3. **Sample cards:** Finally, cards can be sampled based on the selected deck archetype. For this purpose, the agent first removes previously observed cards from the centroids membership sequence. For each observed card, the agent removes the highest value from the centroids membership sequence of this card. The remaining entries in the centroid are ranked according to the sum of their membership sequence. Similar to the bigram-based prediction each card can receive a probability based on the determined sum. The resulting probability distribution can be used to sample cards based on their likelihood to appear in the remaining cluster. For each sampled card the removal process can be repeated to assure that cards are not overrepresented in the resulting prediction set.

5.4 Agent Model

The proposed agent model combines the idea of multiple independent state determinisations of ensemble MCTS [49, 157] with the predictive models proposed in the previous sections. When starting a turn the partially-observable state is used as the root node of the search process. Actions represent transitions to other nodes and are simulated using the game's model. The quality of each node is determined by a simple scoring function that takes the number of cards and minions as well as the health pool of both players into account. This scoring function is a weighted combination of scoring functions for rating aggressive and defensive play, which were initially proposed in the Metastone-framework[8]. A greedy agent is used for fast rollouts during the simulation phase. This agent uses the same scoring function to select its actions.

During the search, the agent implements a 3-phase search process. In the first phase, the actions of the player's turn are optimised using a normal UCT search. Since the cards of the player are unknown at this moment in time, the search is focused on optimising the agent's action sequence until it would be ending its turn. After a fixed number of simulations, the best end-turn nodes are selected for further consideration in the second phase.

At the start of the second phase, the agent utilises one of the proposed card prediction methods to sample the opponent's deck and hand cards according to all previously observed cards. Predicting the opponent's hand cards allows the agent to continue the search process for the opponent's turn. Similar to ensemble MCTS, the search process is split into separate paths of which each path is continued using an independent state determinisation. These are made using a predictive model that can predict likely states based on previously observed cards. In turn, every card observed during the opponent's turn is used to feed the model with new information on the current game-state and improve its prediction accuracy.

After fixing the opponent's turn, the agent is once more trying to optimise the action sequence of its next turn. This step is important to detect actions that should be postponed to the next turn. A simple example is a spell card

[8] https://github.com/demilich1/metastone

that kills all the opponent's minions. This card will receive a high ranking when played during the first turn but could be more useful when used after the opponent played additional minions.

Results of all three phases will be back-propagated along their action-sequences. Finally, the best-rated action will be applied to the game-state. Figure 5.4 depicts the search process of the proposed agent model.

The process is stopped after the third simulation phase. Otherwise, errors in the prediction of the second phase could influence the prediction of upcoming opponent phases. This has shown to reduce the performance of the agent in preliminary evaluations.

5.5 Evaluation of the Prediction Accuracy

Before evaluating the game-playing performance of the developed agent model, the proposed bigram and clustering approaches will be analysed regarding their prediction accuracy. This will be done using public deck data sets of the HSReplay website[9] and replay data of the Collect-O-Bot website[10]. Both sites are part of community services where players contribute their game-playing information to collect and access it centrally.

The deck data set consists of a list of decks being played on the days 5 February 2019 to 20 February 2019. Each record consists of the deck's cards, the number of times the deck has been played, its average win-rate, and its archetype label. The latter is assigned by human experts and will be used for evaluating the results of the proposed fuzzy multiset deck clustering.

The replay data consists of detailed information on recorded matches and is compiled into monthly data sets. Each record includes all observed cards that have been played per match but does not contain any information on the remaining cards of both players' decks. Data of the time-frame 5 February 2019 to 20 February 2019 will be used for training proposed bigram methods. Furthermore, remaining records of the same patch period (21 February 2019 to 3 April 2019) will be used for evaluating the proposed methods' prediction performance. Used data sets, the source code for the following evaluations, and the raw results can be found in the public git-repository [46].

[9] https://hsreplay.net/decks/
[10] http://www.hearthscry.com/CollectOBot

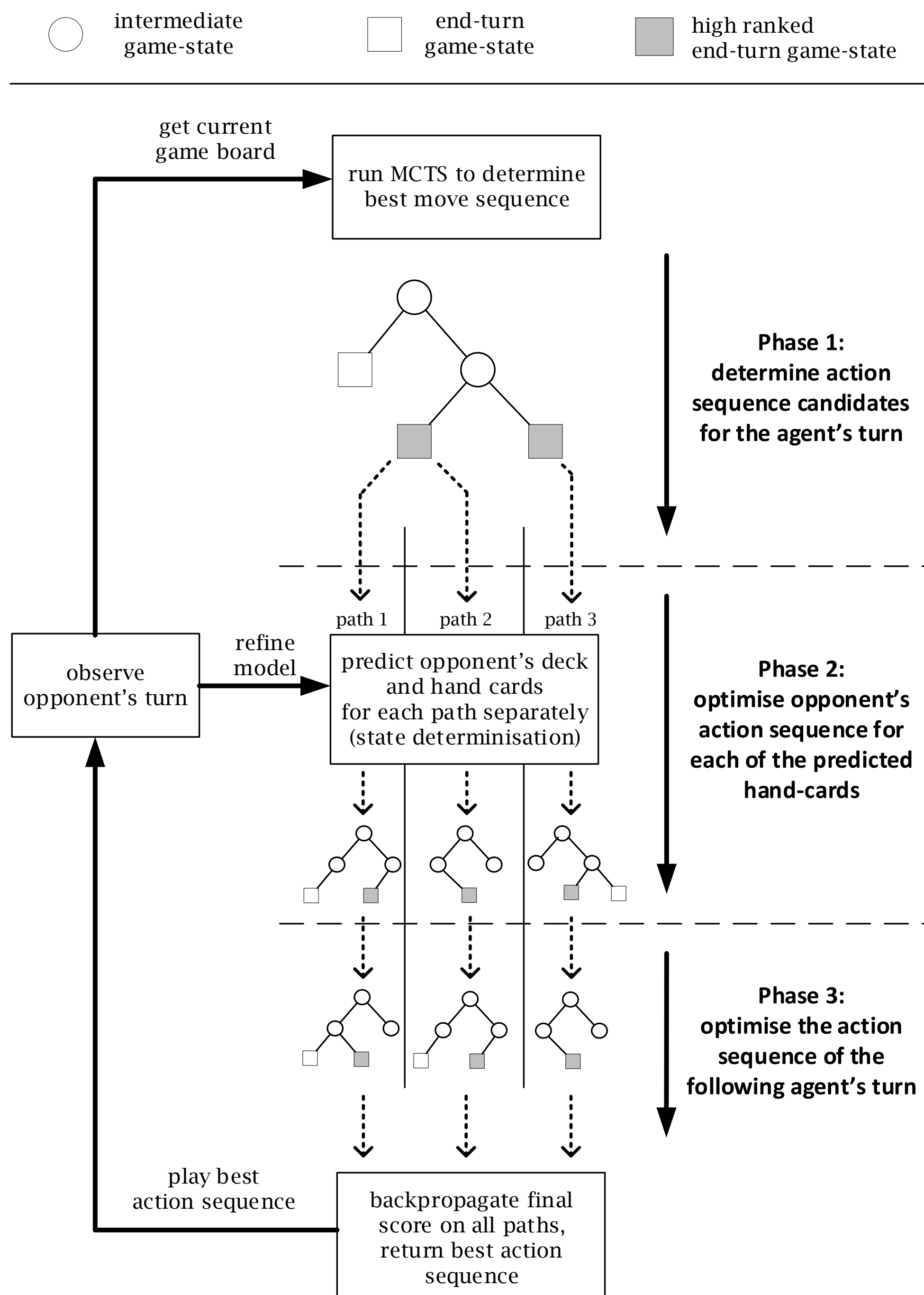


Figure 5.4: 3-phase action-selection using predictive models for state determinisation. During the opponent's turn, the predictive model is updated according to observed actions.

Table 5.2: General contingency matrix

	C_1	C_2	$\cdots$	$C_{K'}$	$\sum$
P_1	n_{11}	n_{12}	$\cdots$	$n_{1K'}$	$n_{1\cdot}$
P_2	n_{21}	n_{22}	$\cdots$	$n_{2K'}$	$n_{2\cdot}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
P_K	n_{K1}	n_{K2}	$\cdots$	$n_{KK'}$	$n_{K\cdot}$
$\sum$	$n_{\cdot 1}$	$n_{\cdot 2}$	$\cdots$	$n_{\cdot K'}$	n

5.5.1 Evaluating the Fuzzy Multiset Clustering

Each of the described clustering algorithms offers a set of parameters that needs to be tuned for optimal results. Since labeled data is available, external validation measures will be used to rate the outcome of each clustering process. External validation measures rate the outcome of the clustering process using external information. Specifically, a set of true labels will be used for comparison with the resulting labels of the clustering process. For this purpose, the measures homogeneity, completeness, and the v-measure will be used in this work.

External Cluster Validation Given a dataset X containing n objects, external validation indices compare the object assignments in the true partition $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ with the assignments of the clustered partition $C = \{C_1, C_2, \ldots, C_{K'}\}$. A contingency matrix counts the number of occurrences n_{ij}, where a point was labeled as cluster C_j and lies in the true partition P_i. Table 5.2 represents a general contingency matrix of the clustering C and the true partition $\mathcal{P}$. For the calculation of external validation measures the probabilities derived from the occurrence counts will be used:

$$p_{ij} = \frac{n_{ij}}{n}; \qquad p_i = \frac{n_{\cdot i}}{n}; \qquad p_j = \frac{n_{j\cdot}}{n} \tag{5.36}$$

Homogeneity ($Hom(C, \mathcal{P})$) measures the extent to which every cluster contains elements of a single class.

$$Hom(C, \mathcal{P}) = \begin{cases} \frac{MI(C,\mathcal{P})}{E(P,P)} & , E(P,P) > 0 \\ 1.0 & , else \end{cases} \tag{5.37}$$

Entropy ($E(\mathcal{P},\mathcal{P})$) has a range of $[0, \log K']$ and Mutual Information ($MI(C,\mathcal{P})$) ranges from $(0, \log K']$. Entropy values close or equal to 0 describe an approximately perfect clustering. The maximal value states that points of a cluster C_i are equiprobable to be in any true partition P_j. Mutual information measures the degree in which two random variables are mutually dependent [14].

$$E(C,\mathcal{P}) = -\sum_i p_i \left(\sum_j \frac{p_{ij}}{p_i} \log \frac{p_{ij}}{p_i} \right) \tag{5.38}$$

$$MI(C,\mathcal{P}) = \sum_i \sum_j p_{ij} \log \left(\frac{p_{ij}}{p_i p_j} \right) \tag{5.39}$$

In contrast to Homogeneity, Completeness ($Compl(C,\mathcal{P})$) measures the extent to which all elements of a given class were sorted into the same cluster.

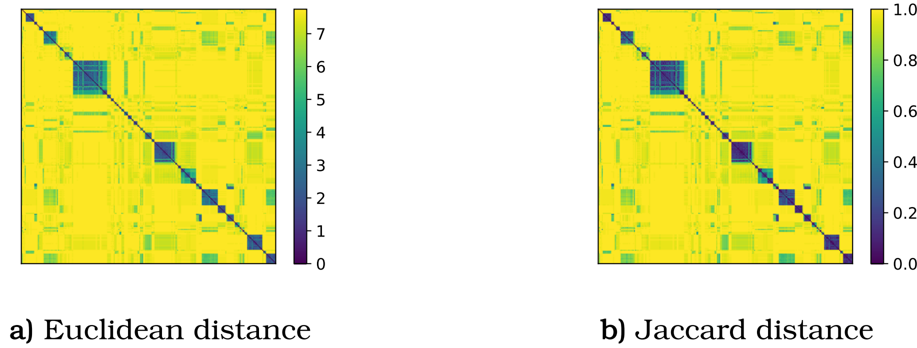
$$Compl(C,\mathcal{P}) = \begin{cases} \frac{MI(C,\mathcal{P})}{E(C,C)} & , E(C,C) > 0 \\ 1.0 & , else \end{cases} \tag{5.40}$$

The V-measure represents the trade-off between Homogeneity and Completeness, by calculating their harmonic mean [146].

$$V(C,\mathcal{P}) = \frac{2 \cdot Hom(C,\mathcal{P}) \cdot Compl(C,\mathcal{P})}{Hom(C,\mathcal{P}) + Compl(C,\mathcal{P})} \tag{5.41}$$

The scores of all three external validation measure fall into the range of [0, 1], whereas 1 represents a perfect clustering result.

Clustering Results The labeled deck data includes 956 deck entries of 72 archetypes recorded during a single patch period. For the clustering process, each deck is represented as (fuzzy) multisets. The distance matrices using Euclidean distance and Jaccard distance are shown in Figure 5.5. Note that the Jaccard distance is bound to the range [0, 1] while the Euclidean distance is non-limited. Comparing the two partial orders implied by these decision matrices reveals that they are the same despite differences in relative values. For this reason, the following experiments will be done using Jaccard distance.



a) Euclidean distance **b)** Jaccard distance

Figure 5.5: Distance matrices for the deck dataset

To compensate for the initialisation problem of k-means, the algorithm is repeated 10 times with varying initialisations for each value of k. The clustering with the best SSE (cf. Equation 5.25) will be reported as the final clustering result. The hierarchical clustering algorithm using single or complete linkage produces a partition hierarchy of which each level represents a clustering of varying sizes. For the algorithms k-means and HAC using either single or complete linkage clusterings for $k = 10 \dots, 250$ clusters are reported. In the case of DBSCAN an initial grid-search is performed to find a suitable parameterisation (see Figure 5.6). It turned out that small values of m_{Pts} are favourable. Furthermore, m_{Pts}DBSCAN [48] will be used to quickly generate a hierarchy of clusters for all relevant parameterisations of the ε-radius. This hierarchy is used to extract the clustering with the most labeled points per number of clusters. The results for $m_{Pts} \in \{2, 5, 10\}$ are reported.

All clusterings are rated using the external validation measures homogeneity, completeness, and the v-measure. Figure 5.7 shows the development of these measures according to the number of clusters. All algorithms quickly increase in completeness during the first merges. In case of DBSCAN using $m_{Pts} \in \{5, 10\}$, discontinuities in the curve can be observed. These can occur when two promising clusters are merged.

The parameters yielding the highest v-measure are summarised in Table 5.3. Reported v-measure values of 0.9 and higher are very promising results. They indicate that the implemented clustering based on fuzzy mul-

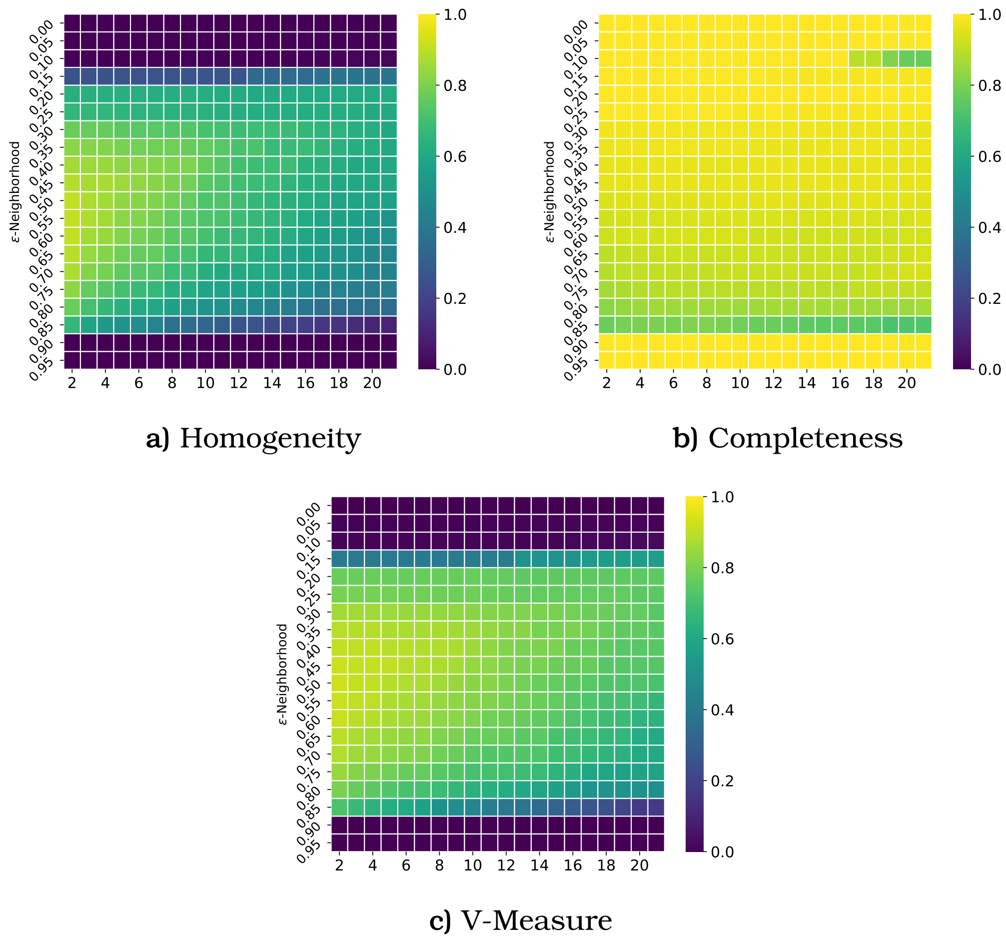
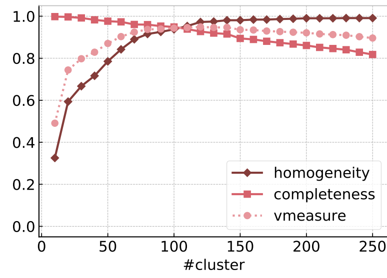


Figure 5.6: Grid-search results for the optimisation of DBSCAN parameters.

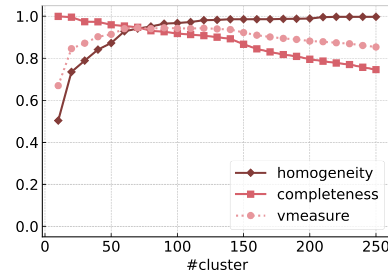
tisets is able to retrieve clusters that match the human expert labels to a large degree. The best performing parameter configurations will be used in the following evaluations of the card prediction accuracy.

Table 5.3: Best performing parameter configuration per clustering algorithm

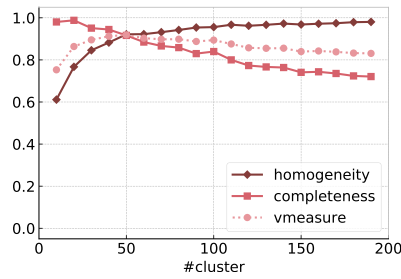
Algorithm	Parameters		Max V-Measure
HAC single linkage	$k = 120$		0.949
HAC complete linkage	$k = 90$		0.945
k-means	$k = 50$		0.919
m_{Pts}DBSCAN	$m_{Pts} = 2$	$\varepsilon = 0.422$	0.923
m_{Pts}DBSCAN	$m_{Pts} = 5$	$\varepsilon = 0.500$	0.905
m_{Pts}DBSCAN	$m_{Pts} = 10$	$\varepsilon = 0.571$	0.870



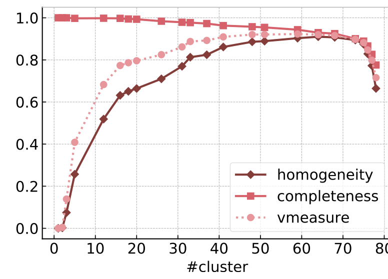
a) HAC single linkage



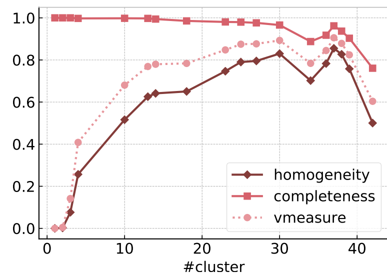
b) HAC complete linkage



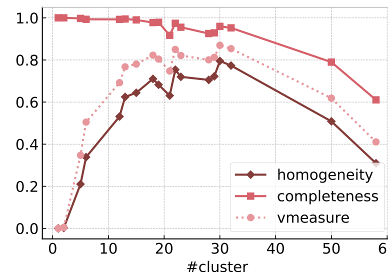
c) k-means



d) DBSCAN $m_{Pts} = 2$



e) DBSCAN $m_{Pts} = 5$



f) DBSCAN $m_{Pts} = 10$

Figure 5.7: Comparison of clustering results based on external validation measures homogeneity, completeness, and the v-measure

5.5.2 Card Prediction Remaining Game

In this first evaluation, the accuracy of predicting upcoming cards of the remaining game will be measured. Since the number of observed cards and the complexity of turns changes over time, results will be reported bucketed by turn. Each algorithm is used to predict the 10 most likely cards after each of the first 10 turns. To assure that the prediction of the last turns can be tested, only games that lasted at least 15 turns have been selected for this evaluation, which results in a total of 3062 games.

Figure 5.8 and Figure 5.10 show the prediction accuracy of proposed methods. The 10 cards which were predicted to be the most likely to appear were evaluated. The highest ranked card is marked green, while the tenth most likely card is marked red and the average accuracy of the ten highest ranked predictions is marked by a blue line. Most curves show that this ranking remains intact throughout the game. However, DBSCAN clustering-based predictions show that cards with a lower rank appear disproportionately often during the first turns. Since this effect vanishes as soon as more cards have been observed (cf. DBSCAN $m_{Pts} = 10$ turns 5-10), these prediction methods seem to require more data. Despite the lack of information, bigram prediction methods perform comparatively well during the first turns with an average prediction accuracy of about 0.5, which is the result of common opening strategies. In contrast, clustering-based prediction methods perform worst in the second turn but increase in accuracy as soon as more cards are observed. Over time, the accumulated knowledge of observed cards suffices to correctly predict the opponent's deck and thus enable the agent to predict upcoming cards with high accuracy. From the sixth turn onwards, all methods show a steady decline in their prediction accuracy which is probably due to the decreasing number turns until the end of the game.

For estimating the opponent's hand cards, a whole set of cards needs to be predicted. Figure 5.9 and Figure 5.11 show the aggregated accuracy of predicted cards. Here, the aggregated accuracy describes the chance that any of the top-k ranked cards will be played. The graphs show the aggregated accuracy of the top-2, top-5, and top-10 predicted cards. Additionally, the prediction accuracy of the highest-ranked card is shown as a baseline. All the proposed methods show a rapid increase in aggregated prediction accuracy when taking multiple predictions into account. This increase becomes smaller when the number of cards to consider is increased, indicating that the highest ranked cards provide the highest value for the prediction. When considering any of the top 10 predicted cards to be correct, the bigram-based prediction methods nearly reach an accuracy of 1.0. Clustering-based prediction methods perform worse during the first turns but increase in accuracy after the second turn. After the fifth turn, every method correctly predicts remaining cards with an accuracy of 0.8 or better.

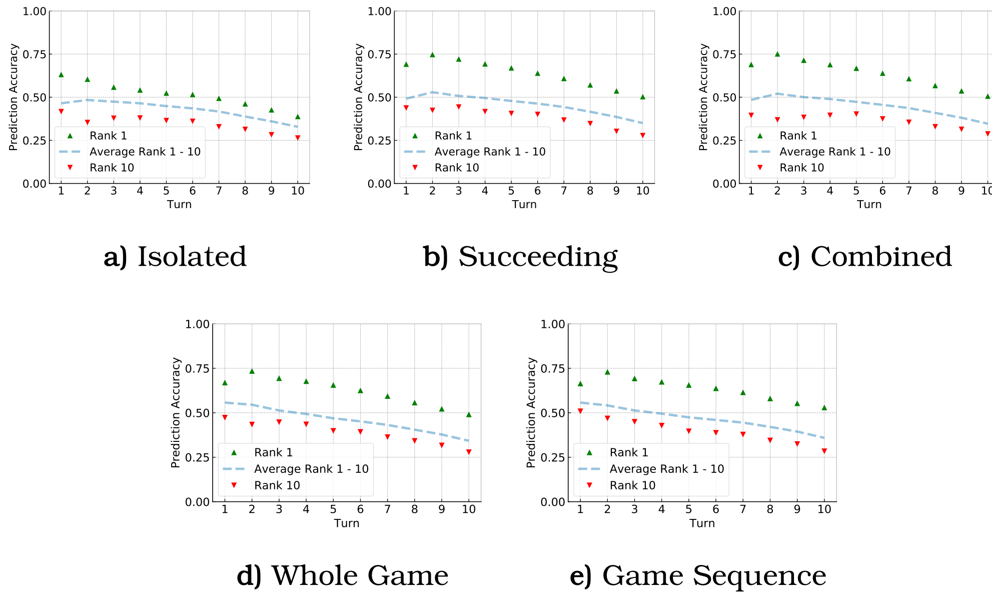


Figure 5.8: Accuracy for predicting cards that may appear in the remaining turns of the game bucketed by turn. Each model a)-e) ranks cards according to their estimated probability of appearance. The validation set is used to determine the accuracy of each prediction for every rank. The highest-ranked card is marked in green, while the prediction of the 10th ranked is shown in red. The average accuracy of all ten ranks is shown in blue.

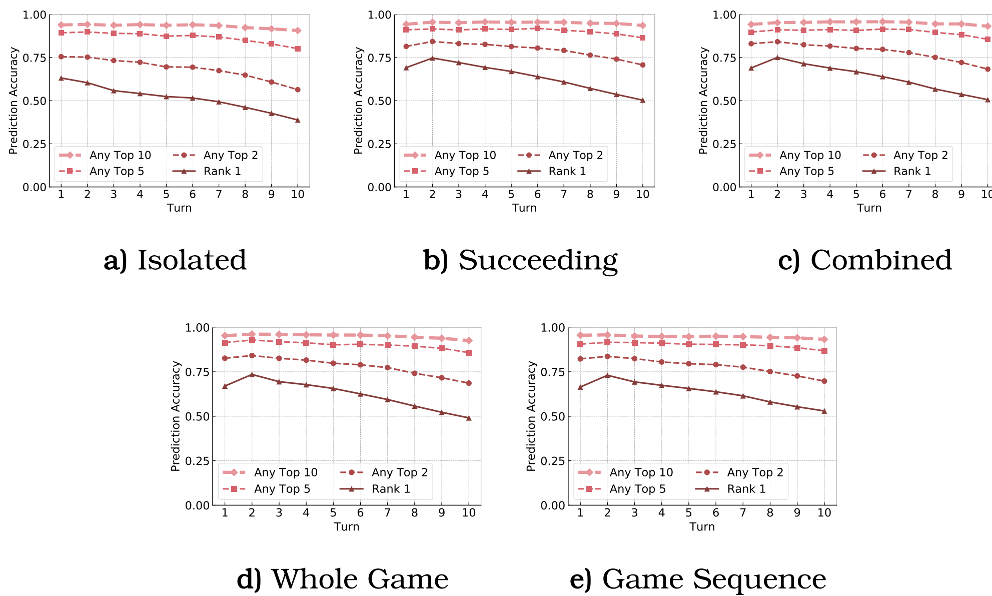


Figure 5.9: Accuracy for the aggregated prediction of cards that may appear in the remaining turns of the game bucketed by turn. Each model a)-e) ranks cards according to their estimated probability of appearance. The validation set is used to determine the accuracy in case the ranks 1-k are considered. Values describe the accuracy of the top k ranked being correctly predicted.

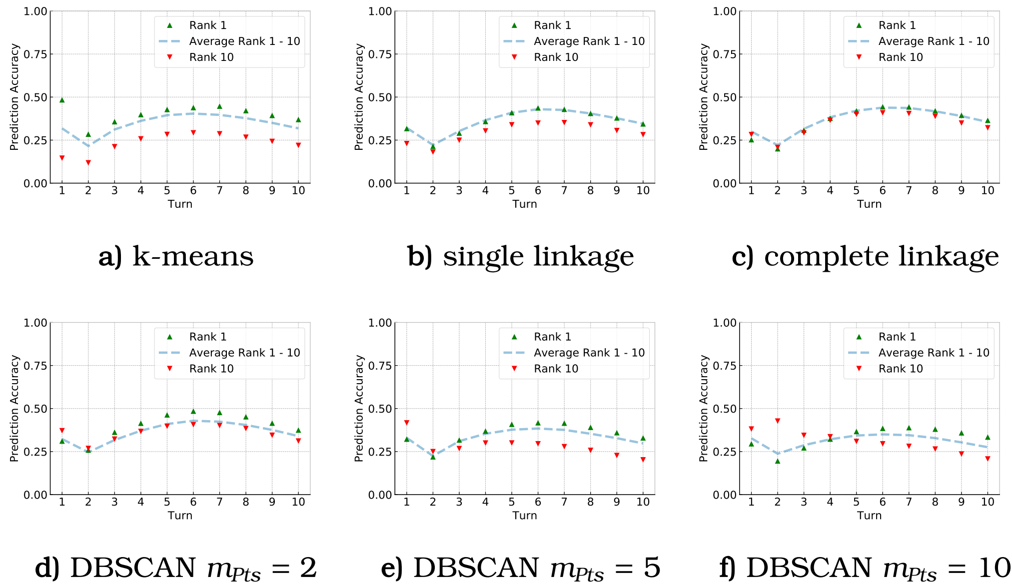


Figure 5.10: Accuracy for the prediction of cards that may appear in the remaining turns of the game bucketed by turn. Subfigures a)-f) show the prediction results based on the clustering result achieved using each algorithm's best performing parameters. The highest ranked card is marked in green while the prediction of the 10th ranked is shown in red. The average accuracy of all ten ranks is shown in blue.

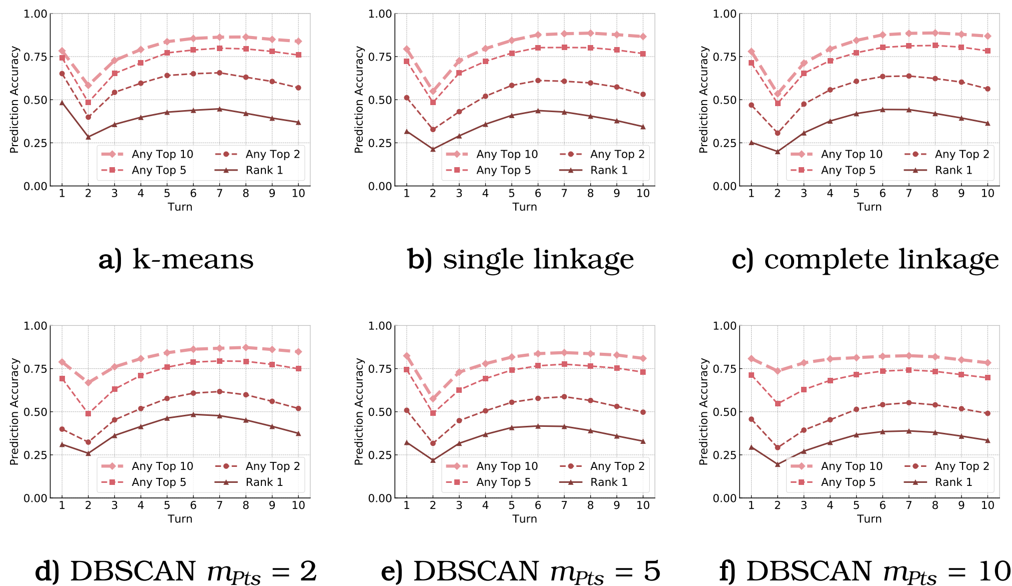


Figure 5.11: Accuracy for the aggregated prediction of cards that may appear in the remaining turns of the game bucketed by turn. Subfigures a)-f) show the prediction results based on the clustering result achieved using each algorithm's best performing parameters. Reported values describe the accuracy of any of the top k ranked cards being correct.

5.5.3 Card Prediction Next Turn

Predicting upcoming cards of the remaining game is useful in case an unlimited search depth is viable. However, due to the many random effects and the variability of decks, the breadth of Hearthstone's game tree is enourmous. Since it is not viable to consider all possible events that can appear till the end of the game, the proposed agent focusses on the optimisation of the next three turns. This is a complex task in itself, since a turn does not consist of a single action but of a sequence of actions. The result of any action sequence depends on the chosen order of actions. After an action sequence has been applied and the opponent's turn has been observed, new information will be available to improve the prediction of the following turns. This second evaluation focuses on the accuracy of predicting upcoming cards of the next turn. For better comparability, the same replay data is used as in the previous evaluation.

Figure 5.12 and Figure 5.14 show the prediction accuracy of proposed methods per rank. Green markers show the prediction accuracy of the highest-ranked card to appear during the next turn. The prediction accuracy of the card ranked tenth is marked in red and the blue line indicates the average prediction accuracy of the first ten ranks. As shown in the diagrams, the accuracy of predicting cards of the next turn is much lower than predicting cards of the remaining game. This is due to the lower number of cards to be observed in the next turn than in the remaining game. Nevertheless, all bigram-based prediction methods (except for the isolated bigram counting) show a promising accuracy of about 25% correct predictions for the highest-ranked card.

In contrast, the clustering-based prediction methods largely fail in predicting cards of the next turn. On average, each of the ten highest ranked predictions reach an accuracy of 10% or lower. Despite their good results in predicting cards of the remaining game, their ranking mechanism cannot effectively be used to predict cards of the upcoming turn. Therefore, these methods may be used to detect the deck of the opponent, but in their current form should not be used to generate a state determinisation in the context of the proposed Hearthstone agent.

Table 5.4: Comparison of the methods' average prediction accuracy in discussed evaluation scenarios.

Algorithm	Single Prediction 1st Rank		Aggregated Top-10	
	full game	next turn	full game	next turn
Isolated	0.515	0.13	0.932	0.506
Succeeding	**0.638**	**0.285**	**0.951**	**0.588**
Combined	0.637	0.274	0.950	0.571
Whole Game	0.622	0.276	**0.951**	0.554
Game sequence	0.633	0.261	0.948	0.543
k-means	0.402	0.045	0.799	0.328
single linkage	0.358	0.044	0.810	0.347
complete linkage	0.362	0.044	0.806	0.349
DBSCAN $m_{Pts} = 2$	0.401	0.050	0.818	0.344
DBSCAN $m_{Pts} = 5$	0.355	0.045	0.788	0.310
DBSCAN $m_{Pts} = 10$	0.330	0.035	0.800	0.294

Similar to the previous evalution, Figure 5.13 and Figure 5.15 show the aggregated prediction accuracy for predicting any of the top-k ranked cards correctly. Once again, the bigram-based prediction methods outperform clustering-based predictions. All bigram-based predictions reach a prediction accuracy of 50% when considering the top 10 ranked cards. This can be very useful when predicting hand card sets of the opponent since the correct card is likely to be included in half of the considered cases. Clustering-based prediction methods steadily improve in accuracy over time. Therefore, cards observed during the first turns seem to be useful in predicting the deck, but not necessarily the upcoming cards of the next turn.

Table 5.4 shows the average accuracy of tested algorithms in all four evaluation scenarios. Overall, cluster-based prediction methods are outperformed by card sequence models. The latter are able to predict cards of the next turn and the remaining game with high accuracy. Among the bigram-based prediction methods the isolated bigram counting performed worse than the other alternatives. The succeeding bigram-counting resulted in the best performance in all four tested scenarios. For this reason, the following evaluation of the proposed agent's game-playing performance will be based on predictions of the succeeding-turn card sequence model.

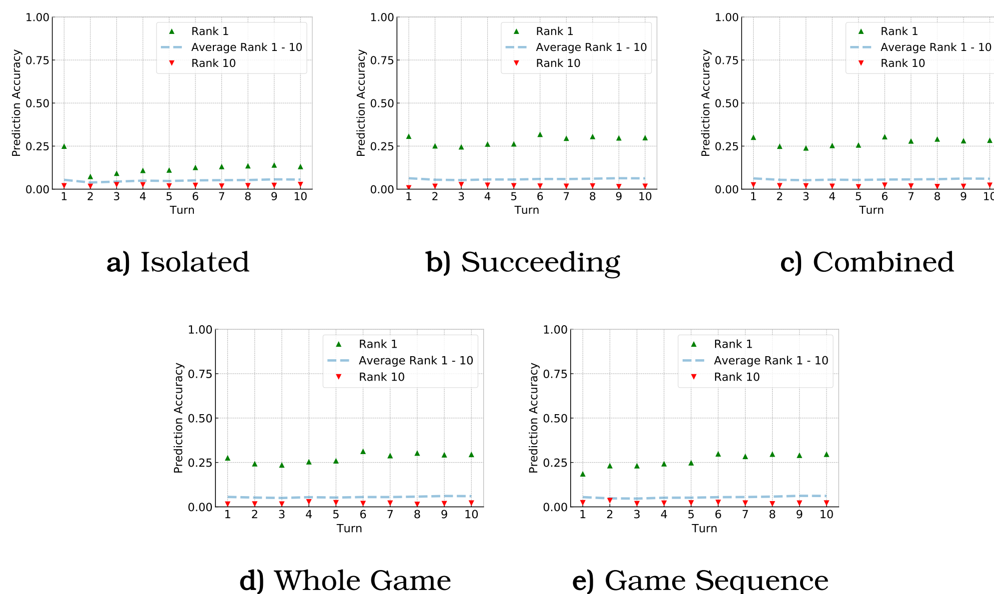


Figure 5.12: Accuracy for predicting cards that may appear in the remaining turns of the game bucketed by the turn the prediction has been made. Each model a)-e) ranks cards according to their estimated probability of appearance. The validation set is used to determine the accuracy of each prediction for every rank. The highest-ranked card is marked in green, while the prediction of the 10th ranked is shown in red. The average accuracy of all ten ranks is shown in blue.

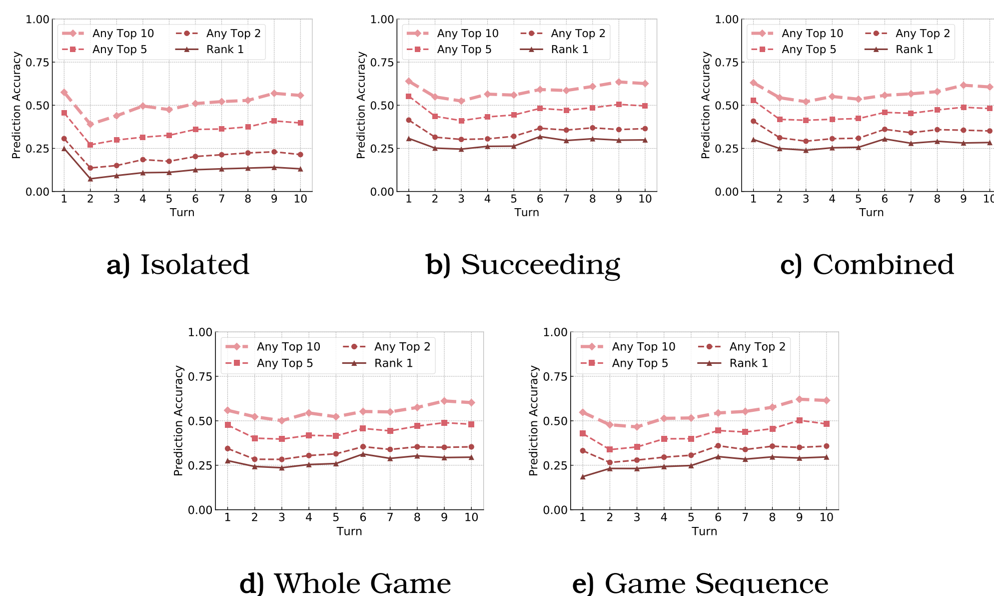


Figure 5.13: Accuracy for the aggregated prediction of cards that may be played by the opponent during its next turn bucketed by the turn the prediction has been made. Each model a)-e) ranks cards according to their estimated probability of appearance. The validation set is used to determine the accuracy in case the ranks 1-k are considered. Values describe the accuracy of the top k ranked being correctly predicted.

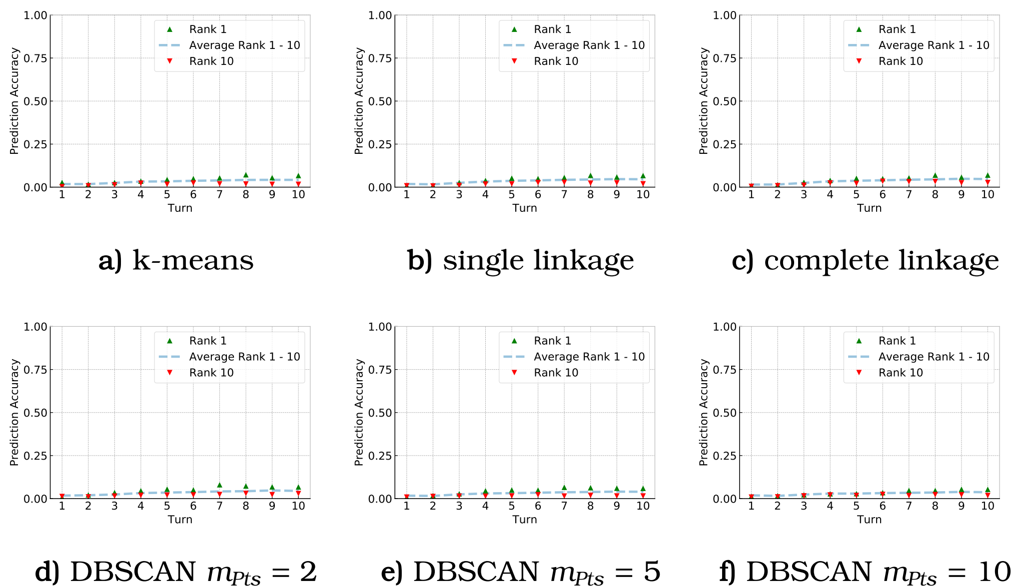


Figure 5.14: Accuracy for predicting cards that may appear in the remaining turns of the game bucketed by turn. Each model a)-f) ranks cards according to their estimated probability of appearance. The validation set is used to determine the accuracy of each prediction for every rank. The highest-ranked card is marked in green, while the prediction of the 10th ranked is shown in red. The average accuracy of all ten ranks is shown in blue.

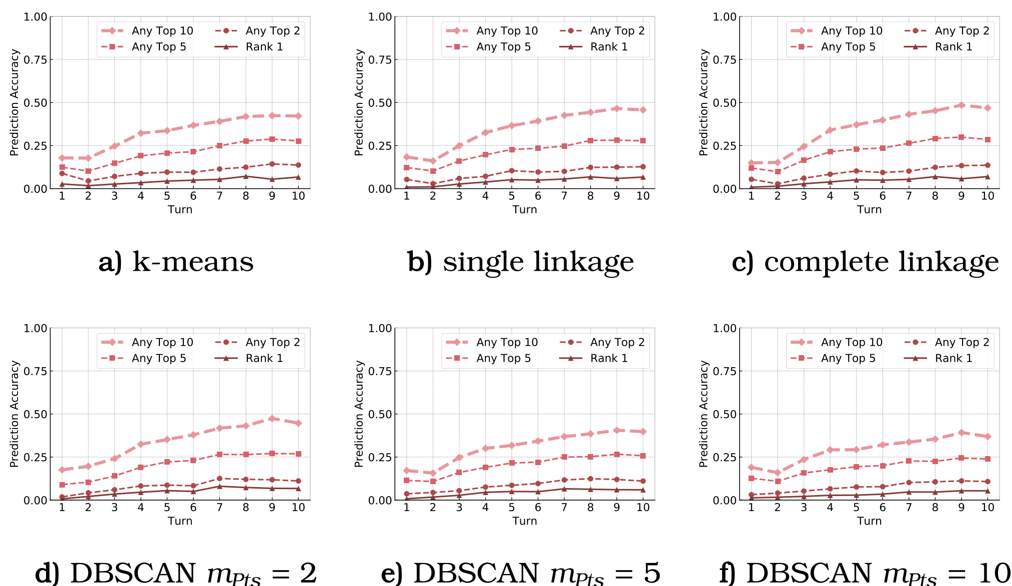


Figure 5.15: Accuracy for the aggregated prediction of cards that may be played by the opponent during its next turn bucketed by the turn the prediction has been made. Each model a)-f) ranks cards according to their estimated probability of appearance. The validation set is used to determine the accuracy in case the ranks 1-k are considered. Values describe the accuracy of the top k ranked being correctly predicted.

5.6 Evaluation of the Game-Playing Performance

It is difficult to compare the proposed agent with other Hearthstone agents found in the literature because they are spread over several frameworks. To overcome this problem, I designed the Hearthstone AI competition [53], an international research competition, based on the community-driven simulator Sabberstone [43]. This competition was part of the 2SS18 IEEE Conference on Computational Intelligence and Games (CIG) and the 2019 IEEE Conference on Games (COG). This framework is written in C# and the competition extends the original framework by multiple helper classes which provide simple means of accessing the current game state limited to variables that would have been observable to a human player.

The agent is given a partially observable representation of the current game-state that is matching the human player's perception. It consists of information about all legally observable parts of the game, namely the game board, the set of remaining cards in the agent's deck, the agent's hand cards, the number of cards in their opponents' hand and a history of all previous events. Each agent is ensured 60 seconds of computation time per turn and the agent can perform multiple actions during a single turn. In case the turn was not ended by the agent, each returned action will be processed irreversibly and an updated game-state will be forwarded to the agent. For this reason, the agent can continuously observe the outcome of executed actions to adjust its next actions. Actions of both players are applied until a winner can be determined or a maximum number of turns (default = 50) is exceeded. In the latter, the game ends with a draw and is restarted until either of the two agents wins. The number of wins, draws, and losses as well as the total and average response times per agent are tracked and reported at the end of a simulation session.

During the first two years, the competition received a total of 80 submissions distributed into two tracks. The pre-made deck playing track consists of a set of known and unknown decks. Submitted agents will be playing against each other. Unknown decks are included such that agents need to be able to adjust to decks they were previously not trained with. In contrast, the user-created deck-playing track allows agents to define the deck they will use during the evaluation. It turned out that the user-created deck playing

track involves a bit of luck when choosing the deck. Since the outcome of a game is not only dependent on the agents' skill, but also their deck choices, an agent that may play better on average can still perform badly in this competition track. For ensuring comparability in-between agents the pre-made deck playing track's competition mode will be used.

5.6.1 Evaluation Setting

The proposed agent will compete against a variety of top-performing agents of the 2018 and 2019 competition. Agents were selected based on performance in their respective competitions and the algorithm they represent. Note that an early version of the proposed agent scored first place in the 2018's competition and should, therefore, not be considered as an opponent in this evaluation. Additionally, the second place of the 2018's competition was resubmitted in an extended version in 2019 such that only the updated version will be considered.

The set of agents includes the following bots:

- **Tyche Agent (MCTS) by Kai Bornemann:** the agent uses monte carlo tree search to optimise the action sequence of its current and upcoming turns. The opponent's turn is not considered during the search process. The agent uses a different scoring heuristic for each deck. It ranked third place in the 2018's competition.

- **Tyche Agent (One-Step Lookahead) by Kai Bornemann:** the agent is similar to its MCTS version but does only search for the optimal action sequence of its turn. The agent ranked fourth place in the 2018's competition

- **Alpha-Beta Pruning Agent by Hans-Martin Wulfmeyer:** the alpha-beta pruning agent optimises the action sequence by considering its own and its opponent's turn. It ranked fifth place in the 2018's competition.

- **Greedy Agent by Ivan Prymak and Milena Malysheva:** this greedy agent utilises a hand-tuned scoring function and focuses on optimising the order of attacks. For this reason, the agent is very efficient in playing aggro decks. The agent ranked sixth place in the 2018's competition.

- **Pruned Breadth-First Search by Tom Heimbrodt:** the best performing agent in the 2019's competition utilised a pruned breadth-first search with a custom heuristic and variable search depth. The heuristic strongly favours retaining control over the board by keeping the number of minions and attack points of the opponent's minions as small as possible. It was the result of an evolutionary optimisation process on all hero classes including the three publick decks and other popular decks of past patch periods.

- **Álvaro Agent (MCTS) by Álvaro de Marcos Alés:** the agent combines MCTS with an evolutionarily optimised scoring function [67]. It ranked second place in the 2019's competition.

- **Beam Search Agent by Daniel Bokelmann and Malte Unkrig:** beam search [144] is a heuristic search algorithm that only expands the most promising nodes at each layer. The agent reached the fourth place in an internal evaluation, but barely missed the final round in 2019's international competition.

- **Greedy Agent by Lars Wagner:** this is an updated version of the 2018's Jade Druid agent by Lars Wagner, which was the winning entry of the User-Created Deck Playing track. It features an evolutionary optimised scoring function and implements expert rules in its greedy decision function.

The following evaluation will make use of the same six decks as in the competition of 2019. At the time of development, only the first three decks were publicly announced. These three decks were also used in the 2018's competition. The remaining decks were unknown at the time of submission. Each of these decks uses another hero and requires a different play-style:

- **Aggro Pirate Warrior:** this aggressive deck lets the agent play many minions and weapons to defeat the opponent as fast as possible. The synergies among pirate minions allow for a fast-paced playstyle but the deck is missing stronger options in late turns.

- **Midrange Jade Shaman:** the midrange jade shaman makes frequent use of the jade golem mechanic, which spawns increasingly stronger

minions over time. Therefore, this deck may struggle against aggressive decks during the early-game but can provide the player with very strong options during the late-game.

- **Reno Kazakus Mage:** this control deck consists of many spell cards that allow the removal of smaller minions from the board. Later options include strong minions and damage spells to defeat the opponent in the late-game.

- **Midrange Buff Paladin:** this deck allows agents to play many weak minions and use a variety of spells to increase their health and attack power. The deck concentrates on efficient minion attacks while constantly keeping control over the board.

- **Miracle Pirate Rogue:** the miracle pirate rogue plays similar to aggro pirate warrior during the early-game. However, the combo mechanic of the Rogue-hero allows the agent to chain multiple cards in the same turn to increase their efficiency. This unique mechanic requires the agent to plan multiple steps ahead for detecting each card's potential.

- **Zoo Discard Warlock:** this deck is built to make frequent use of the Warlock's hero power, which allows the player to draw a card at the cost of 2 mana and 2 life points. In combination with unique minion effects, this can allow the player to quickly draw cards, play minions, and gain control over the board. Nevertheless, investing life for drawing cards can be a risky, but also rewarding strategy.

The proposed agent competes against the eight aforementioned agents in a round-robin tournament. A match-up between two agents consists of 360 games. In these, each combination of decks is played 10 times, whereas in half of these games the first agent will be the starting player. The proposed agent is configured to use the succeeding bigram counting scheme since it represents a trade-off between high-quality predictions and low memory consumption. Its scoring function is optimised for the first four decks to validate the influence of the scoring function on the agent's game-playing performance.

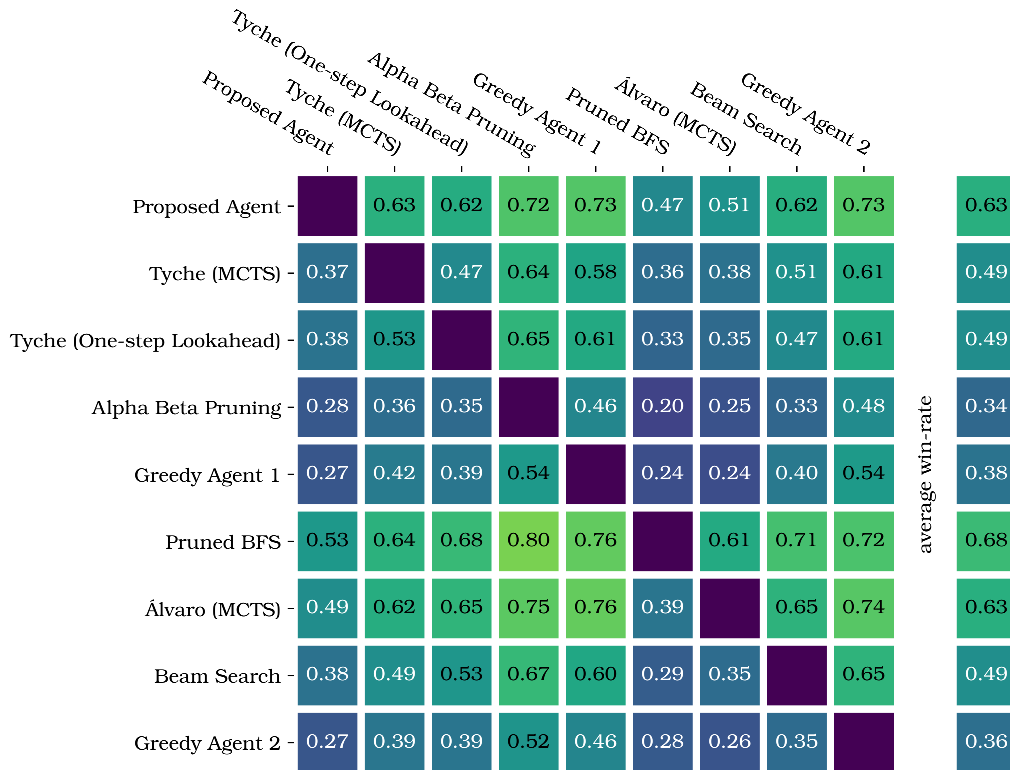
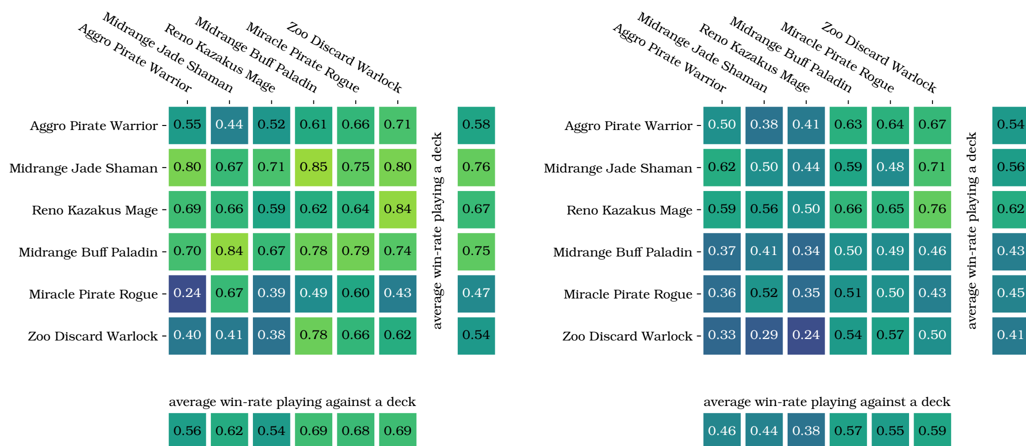


Figure 5.16: Win-rate for each match-up after 360 simulated games. The cell (i,j) indicates the win-rate of agent *i* against agent *j*. The average win-rate per agent is shown on the right.

5.6.2 Results

Figure 5.16 shows the average win-rate of each match-up and the average win-rate per agent. The proposed agent is able to win on average against all agents except for the pruned BFS by Tom Heimbrodt, which is the agent with the highest average win-rate. The MCTS agent by Àlvaro performed similarly to the proposed agent. Both Tyche agent variants and the Beam search agent win nearly the same amount of games as they lose. In contrast, both greedy agents, as well as the Alpha-Beta Pruning agent, performed worst in the evaluation with an average win-rate below 40%. Detailed results of the proposed agent against each of the other agents are shows in Figure 5.19.

When designing the scoring function of the proposed agent, only the first four decks were considered. However, the Miracle Pirate Rogue and



a) Proposed agent's average win-rate per deck while playing against all other agents.

b) Average win-rate per deck based on all games played during the evaluation.

Figure 5.17: Average win-rate per combination of decks. Each cell (i, j) describes the win-rate of the deck in row i against the deck in column j. Average win-rates for playing a deck and playing against a deck are shown on the right and the bottom of each figure.

the Zoo Discard Warlock decks both include unique mechanics that may be misrepresented in the trained scoring function. Win-rates of the proposed agent per deck were calculated to test if the combination of decks and the trained scoring function resulted in reduced performance. Figure 5.17a shows the win-rate of the proposed agent playing against all other agents for each combination of decks. The average win-rates per deck show that the two decks that were not considered during the training of the scoring function resulted in the lowest win-rate per deck.

In comparison, Figure 5.17b shows the average win-rate of all agents per combination of decks. Based on this win-rate it is possible to identify advantageous deck match-ups and the general strength of decks based on their average win-rate. The midrange buff paladin deck, the miracle pirate rogue deck as well as the zoo discard warlock underperform in this evaluation setting. Nevertheless, the proposed agent's win-rate using these decks is higher than the average win-rate of all agents. This result shows

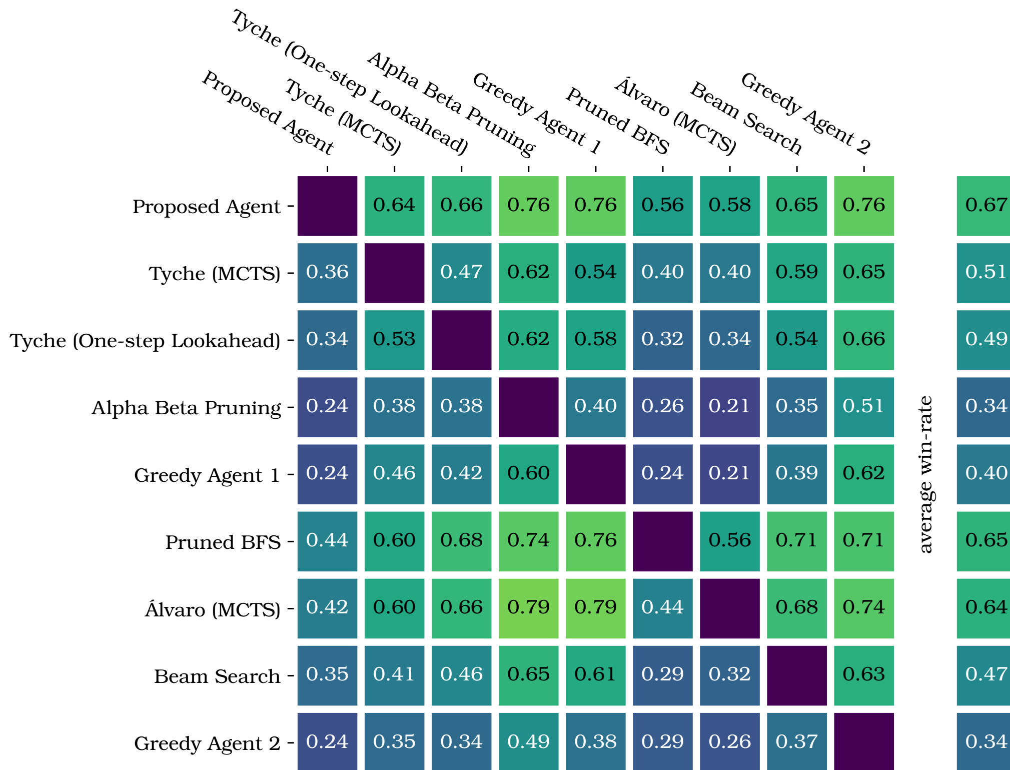


Figure 5.18: Matrix showing the win-rate for each match-up when only considering decks the proposed agent was trained for. The inclusion of the first four decks results in a total of 160 simulated per match-up. The average win-rate per agent is shown on the right.

that the search scheme can still enable the proposed agent to outperform other agents, despite the non-optimal scoring function. Further comparing the agent's win-rate per deck with the average win-rate per deck also reveals some deck combinations in which the agent performs worse than the average. The biggest performance difference can be seen when the proposed agent plays the Miracle Pirate Rogue deck against the Aggro Pirate Warrior deck. This could be attributed to the scoring function.

To see if the proposed agent's performance increases when only considering decks to which the scoring function was optimised, the match results were filtered for matches that only consisted of the first four decks on which the agent was trained on. Figure 5.18 shows the averaged results for each

match-up when only considering this subset of decks. Here, the proposed agent is the best performing among all tested agents with an average win-rate of 67%. Now, the pruned BFS is the second-best performing agent with an average win-rate of 65% closely followed by the Àlvaro agent with a win-rate of 64%. In general, the average win-rate of other agents' seem to be changing by only 1-2%.

The inclusion of multiple MCTS-based agents allows for a comparison of their differences. The Tyche agent is using a standard metastone scoring function and implements no further search optimisations. In contrast, the Àlvaro agent uses an optimised scoring function for all hero classes but does not include further search optimisations. The proposed agent incorporates an optimised search scheme by predicting the opponent's moves and allowing for an increased search depth. Additionally, it features a scoring function optimised for some of the evaluation decks. While the Tyche agent performed worst among these agents, the results of the Àlvaro agent as well the proposed agent show that an increase of the search depth, as well as the optimisation of a scoring function, can improve the overall quality of the agent. The experiments have shown that implementing a predictive scheme for state determinisation or optimising the scoring function can result in a similar performance. Nevertheless, an agent that is implementing both, as seen with the proposed agent in the filtered result table, shows that further improvements in game-playing performance can be achieved.

In a preliminary study [56], we focused on the evaluation of the predictive performance of the proposed agent and its influences on its game-playing performance. Results have shown that the proposed agent outperforms MCTS agents using a random state determinisation with an average win-rate of 55-68%. Additionally, the agent was able to achieve a 46-51% average win-rate against an MCTS agent that was aware of its opponent's cards. Further improving the accuracy of the prediction may push this closer to an average win-rate of 50%.

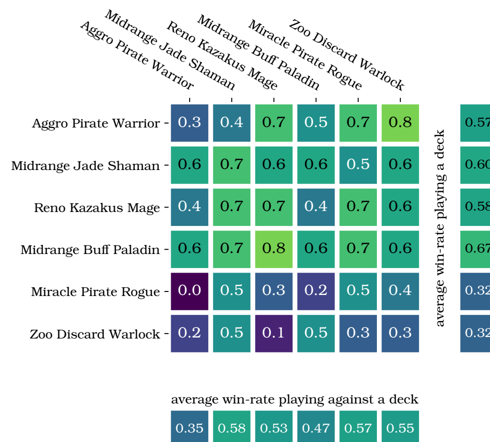
In summary, the results of these experiments indicate that the proposed agent model can be used to handle imperfect information states. The proposed methods for predictive state determinisation can predict upcoming cards with high accuracy and, therefore, allow the agent to increase its search depth.

a) Tyche (MCTS) **b)** Tyche (One-Step Lookahead)

c) Alpha Beta Pruning **d)** Greedy Agent 1

Figure 5.19: Proposed agent's average win-rate per deck while playing against a single opponent.

e) Pruned BFS



f) Àlvaro (MCTS)

g) Beam Search

h) Greedy Agent 2

Figure 5.19: Proposed agent's average win-rate per deck while playing against a single opponent.

CHAPTER 6

Conclusion and Future Work

In this thesis, two variants of prediction-based search processes were proposed, namely forward model learning and predictive state determinisation. The forward model learning framework covers scenarios in which the forward model is unknown by learning a substitute model, which can further be used during action-selection. In contrast, predictive state determinisation improves the way in which games can be played in case the state cannot be fully observed by predicting a subset of likely game-states. Both methods require a training process in order to adapt the prediction models to the respective task. Proposed frameworks were tested on recent game benchmarks in which they have shown to outperform state-of-the-art algorithms.

The forward model learning framework was evaluated by testing the agent's game-playing performance in 30 games of the GVGAI framework. In case a pre-trained model is used, both, the local and the object-based forward model, have shown to outperform a random agent in most of the tested games. Scenarios in which the trained models were inaccurate did not perform well since the agent is not able to recognise good actions based on the predicted state. This limits the agent in a transfer-learning scenario, where played levels include previously unseen scenarios. However, tests with continuous learning have shown that the agent is able to quickly adapt to these scenarios by updating the model accordingly.

The predictive state deterministation method was evaluated using the card game Hearthstone. By comparing the proposed agent to agents submitted to the Hearthstone AI competition it was shown that a search based on predictive state determinisation yields better game-playing performance.

6.1 Discussion and Research Questions

To summarise this thesis, I will shortly discuss its results with respect to the research questions initially posed.

Review of the State-of-the-Art

Q1 **Which methods exist and are applicable in case the agent cannot access a game's forward model or fully observe its current state?**

In Chapter 3 state-of-the-art algorithms for implementing autonomous game-playing agents have been reviewed. In case the agent cannot access a forward model it is limited to the analysis of the current state. Heuristics, either implemented by the agent's developers or generated by an evolutionary approach, can be used to play any game. However, their results have been overshadowed by reinforcement learning and search-based approaches. While the former can be used in the absence of a forward model, they require a large number of training examples to reach satisfying performance. In contrast, search-based approaches are known to perform well without any training, but strictly require the environment's forward model. Since the success of reinforcement learning algorithms has already show that it is possible to learn the reward of an action, the same could be done for predicting its result. This idea became the main motivation of prediction-based search approaches, which were implemented with proposed forward model learning techniques in Chapter 4.

If the state cannot be fully observed, heuristics are still applicable, but are limited to the processing of the partial state observation. The same applies to reinforcement learning algorithms, which have shown to be successful in learning games such as Poker, but require long training times. The two simulation-based search algorithms, ensemble-UCT and information set MCTS, have been implemented to handle similar situations. However, due to their uniform state sampling, they have been shown to perform worse than non-determinising algorithms in case of large state spaces. To overcome this problem the predictive state determinisation method has been proposed in Chapter 5.

Forward Model Learning

Q2.1 Which characteristics can be used to compare forward model learning processes and their results?

A general analysis of forward models has been presented in Section 4.1. Here, the similarity of forward models and classifiers have been highlighted. Several characteristics for the evaluation of forward model learning algorithms have been presented and include accuracy, processing speed, generalisability, learning speed, and interpretability.

Q2.2 How can a forward model be learned by observation of the agent's interaction with its environment, and how can the model be represented and learned efficiently? To which degree do the proposed models fulfil these criteria?

Four forward model types have been presented in this thesis, namely the end-to-end forward model, the decomposed forward model, the local forward model and the object-based forward model. The end-to-end forward model represents the direct mapping of the current state and action to the upcoming state. The decomposed forward model introduces independence assumptions for observed sensor values to model each transition independently. In case each sensor shares the same semantics and the state is represented in a grid-like structure, the local forward model can be applied. It introduces the concept of local neighbourhoods to define the input of local transition functions, which can be applied to any sensor. In contrast, the object-based forward model makes use of additional high-level knowledge on the state representation to further compress the learned forward model.

A qualitative comparison based on the introduced criteria shows that the local forward model and the object-based forward model are suited best for the task of forward model learning in case their requirements are met (cf. Section 4.6). Otherwise, the decomposed forward model can be efficient in representing an environment's forward model but requires the agent to train a separate model per observable sensor

value. In case these three forward model types are inapplicable, the end-to-end forward model could be used, but this comes at the cost of reduced interpretability and generalisability.

Q2.3 **How can the accuracy of forward model learning approaches and their resulting game-playing performance be evaluated? Which of the proposed models performs best?**

The proposed methods have partly been evaluated in previous studies, in which it was shown that they are able to replicate environment models. Nevertheless, the suitability of the end-to-end and the decomposed forward models are limited to less complex environments due to their large hypothesis spaces. Since the local and the object-based forward models have already proven their worth in more complex problems, they are further evaluated in the context of this thesis.

For this purpose, a prediction-based search was implemented using one of the search algorithms BFS, RHEA, or UCT and combining it with the trained forward models. The model's accuracy and the agent's game-playing performance have been evaluated in a general game-learning task including 30 games of the GVGAI framework. Three training setups have been compared, namely training a constant model, continuously updating a model, and transfer learning. Results show that all of the prediction-based search agents overall outperform a random agent when using a pre-trained model. In the few exceptions, either the training data set has not been sufficient or critical components have not been represented in the model's input. Continuously updating the model has shown to quickly increase its accuracy over time while allowing the agent to improve its game-playing performance in just a few trials. The transfer learning evaluation has shown that agents may struggle in case their forward model is not representative for the validation levels. This raises the question of how suitable training level sets can be arranged to support the agent's learning process.

Overall, the local forward model performed best in the pre-trained model and the continuously learning evaluation. Therefore, the model seems to be able to represent an environment's model well enough to play a range of games with good game-playing performance in case the

training data set is representative. In a transfer learning setting agents' using a local forward model were outperformed by agents' using an object-based forward model. This may indicate that the latter is more robust in previously unseen situations.

State Determinisation

Q3.1 **How can the probability of each state be determined and how should a state be sampled?**

The analysis at the beginning of Chapter 5 has shown that it is impractical to store the probability of each state in large state spaces. In particular, the deck construction of deck building games leads to very large state spaces, which would require a database of even more games to appropriately estimate the probability of each state. However, the task can be simplified by focussing on the subset of probable states which are most relevant for the agent's subsequent search. This was done by analysing the current meta-game and exploiting the clustered deck space. For this purpose, two methods have been proposed, namely the card sequence model (Section 5.2) and a clustering-based model (Section 5.3). The former aggregates the information gained by the sequence of previously observed cards to predict the next card of this sequence. The second method uses a fuzzy multiset clustering to add a layer of abstraction to the prediction process. Here, the agent first predicts the probability of each deck to further sample card sets of the most probable decks. To maximise the accuracy of this approach, parameters of used clustering algorithms have been tuned to match human expert labels.

Q3.2 **Can the performance of state determinisation-based search methods be improved by the application of predictive models?**

The card sequence model and the cluster-based model have been compared with respect to their prediction accuracy for upcoming cards (Section 5.5). Both methods have shown that they are able to predict cards of the next turn and cards of the remaining game with high accuracy. Since these models can be used to predict a deck of

cards, the aggregated prediction accuracy over the highest ranked cards was used as a second performance indicator. Results show that the proposed methods increase in accuracy over time, despite the increasing complexity of the players' turns. The sampling process based on the proposed card sequence models resulted in the highest accuracy for predicting the opponent's next card and was further used in testing its influence on the agent's game-playing performance.

Q3.3 **How does the resulting agent's performance compare to the state-of-the-art?**

The proposed agent has been compared against eight of the best-performing submissions of the Hearthstone AI competition (Section 5.6). This competition has been created to build a representative collection of state-of-the-art algorithms for simulation-based search in collectible card games.

Comparing the agents' average win-rate shows that the proposed agent performed second-best in the evaluation. However, reviewing the proposed agent's average win-rate per matchup shows that it performs worst when playing against decks that were not represented in the card sequence model's training data set. When considering all match-ups for which the training data set can be considered representative, the agent performed better than all other tested agents.

By comparing tested agents and their used search schemes, it becomes evident that the proposed predictive state deterministation yields further performance increases over other simulation-based search agents. Comparing the performance of predictive state deterministation against a uniform sampling approach has shown that the former outperforms the latter with an average win-rate of 55%-68%, depending on the current combination of decks.

6.2 Future Work

Proposed frameworks for prediction-based search provide many interesting opportunities for further improvements of the prediction accuracy and the agent's game-playing performance. Since these methods strictly divide

the model into two building blocks, namely the prediction algorithm and the search algorithm, further performance increases may be achievable by replacing common algorithms used in this work with more optimised alternatives. The main focus of this thesis was the design of this two-part process which has shown great performance in chosen evaluation settings. Based on the insights of these experiments, further improvements may be achievable by studying the following topics:

Forward Model Ensembles and Forward Model Fusion Even if the proposed methods for forward model learning have been evaluated independently, agents may also train multiple models at the same time. By keeping track of each model's accuracy, the agent could decide which model to apply in the given situation. Alternatively, a forward model ensemble could be created by aggregating the predictions of each model. In contrast, data fusion approaches could be used to build more powerful models by the inclusion of different data sources, such as visual and object-based information.

Curriculum Learning The evaluation of the trained forward model's transfer learning abilities has shown that they are limited in case the training levels were not representative of future test scenarios. Studying which aspects need to be represented in the training data may allow game designers to gain further insights into the game's design process. Furthermore, creating a series of increasingly difficult levels may ease the learning process of both the player and the forward model learning agent.

Active Learning The proposed forward model learning framework would be an excellent application for active learning. In the context of games, this would require the agent to query scenarios that have not been sufficiently represented in the training data. In case the devised prediction model is able to estimate the confidence of its prediction, the agent could favour scenarios in which the model's confidence is still low. This could reduce the training time while increasing the model's confidence.

Risk Evaluation Evaluating the risk of the agent's action selection can also be based on the idea of measuring the agent's confidence in its prediction of

the next state. Considering the likelihood of alternative outcomes may allow the agent to identify risky actions which should be avoided until the agent's confidence in its forward model has increased.

Transfer learning across games While generalisability has already been discussed as one qualitative criterion of forward models, the generalisation across multiple games has not been focused on yet. In her work "Project Thyia: A Forever Gameplayer", Gaina has proposed an agent that continuously learns to play games [64]. The continuous learning aspect has already been discussed in this thesis. However, the current model representation would need to be adjusted to generalise models across multiple games. Training models that would apply to a whole genre of games, e.g. by modelling typical behaviours of characters and objects, will be an interesting next step in forward model learning.

Other applications In this thesis, it was shown that the agent's performance can benefit from prediction-based search in the context of games. Due to the generality of the forward model learning and the predictive state determinisation frameworks, it would be possible to transfer them to other applications. In terms of forward model learning, the field of robotics would be an interesting application area. Here, robots are often provided with a physics model to plan how a certain position can be reached or an object can be grasped. However, the model often fails to represent objects of the robot's environment. By using forward model learning the robot may be enabled to model other objects and how their state is influenced by the robot's actions.

Bibliography

[1] A. M. Alhejali and S. M. Lucas, "Evolving diverse ms. pac-man playing agents using genetic programming", in *2010 UK Workshop on Computational Intelligence (UKCI)*, Sep. 2010, ISBN: 9781424487752 (cited on page 32).

[2] ——, "Using genetic programming to evolve heuristics for a monte carlo tree search ms pac-man agent", in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 2013, ISBN: 9781467353113 (cited on page 40).

[3] C. F. Aliferis, A. Statnikov, I. Tsamardinos, S. Mani, and X. D. Koutsoukos, "Local causal and markov blanket induction for causal discovery and feature selection for classification part I: Algorithms and empirical evaluation", *Journal of Machine Learning Research*, vol. 11, pp. 171–234, 2010, ISSN: 15324435 (cited on page 60).

[4] D. Anderson, C. Guerrero-Romero, D. Perez-Liebana, P. Rodgers, and J. Levine, "Ensemble Decision Systems for General Video Game Playing", in *2019 IEEE Conference on Games (CoG)*, London: IEEE, Aug. 2019, ISBN: 9781728118840 (cited on page 42).

[5] P. Androulakakis and Z. E. Fuchs, "Evolution of kiting behavior in a two player combat problem", Aug. 2019 (cited on page 32).

[6] D. Apeldoorn and V. Volz, "Measuring strategic depth in games using hierarchical knowledge bases", Aug. 2017 (cited on page 80).

[7] D. Ashlock, D. Perez-Liebana, and A. Saunders, "General video game playing escapes the no free lunch theorem", in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, Aug. 2017, pp. 17–24, ISBN: 978-1-5386-3233-8 (cited on page 42).

[8] I. Azaria, A. Elyasaf, and M. Sipper, "Evolving Artificial General Intelligence for Video Game Controllers", in *Genetic Programming Theory and Practice XIV*, Springer International Publishing, 2018, pp. 53–63, ISBN: 9783319970882 (cited on page 32).

[9] J. Bach, “MicroPsi 2: The Next Generation of the MicroPsi Framework”, in *Artificial General Intelligence*, J. Bach, B. Goertzel, and M. Iklé, Eds., vol. 7716, Springer Berlin Heidelberg, 2012, pp. 11–20, ISBN: 9783642355059 (cited on page 1).

[10] ——, “Modeling Motivation in MicroPsi 2”, in *Artificial General Intelligence*, vol. 9205, Springer International Publishing, 2015, pp. 3–13, ISBN: 9783319213644 (cited on page 1).

[11] H. Baier and P. I. Cowling, “Evolutionary MCTS for Multi-Action Adversarial Games”, in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, vol. 2018-Augus, IEEE, Aug. 2018, pp. 1–8, ISBN: 9781538643594 (cited on page 4).

[12] C. Ballinger and S. Louis, “Learning robust build-orders from previous opponents with coevolution”, in *2014 IEEE Conference on Computational Intelligence and Games*, IEEE, 2014, ISBN: 9781479935468 (cited on page 32).

[13] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, “Speeded-Up Robust Features (SURF)”, *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346–359, Jun. 2008, ISSN: 10773142 (cited on page 73).

[14] J. Bellamy, T. M. Cover, J. a. Thomas, and R. L. Freeman, *Elements of Information Theory Telecommunication Transmission Handbook*, 3rd ed. John Wiley and Sons, Inc., 1991, ISBN: 0-471-20061-1 (cited on page 125).

[15] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The Arcade Learning Environment: An Evaluation Platform for General Agents”, *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013 (cited on page 34).

[16] A. Benbassat and M. Sipper, “EvoMCTS: Enhancing MCTS-based players through genetic programming”, in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 2013, ISBN: 9781467353113 (cited on page 40).

[17] H. Berliner, “Backgammon program beats world champ”, *ACM SIGART Bulletin*, no. 69, pp. 6–9, Jan. 1980 (cited on page 24).

[18] M. R. Berthold, C. Borgelt, F. Höppner, and F. Klawonn, *Guide to Intelligent Data Analysis*, ser. Texts in Computer Science. London: Springer London, 2010, p. 399, ISBN: 978-1-84882-259-7 (cited on pages 52, 85).

[19] O. Biran and C. Cotton, "Explanation and Justification in Machine Learning: A Survey", *IJCAI Workshop on Explainable AI (XAI)*, pp. 8–14, 2017 (cited on page 75).

[20] R. Bjarnason, A. Fern, and P. Tadepalli, "Lower bounding klondike solitaire with monte-carlo planning", in *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'09, Thessaloniki, Greece: AAAI Press, 2009, pp. 26–33, ISBN: 978-1-57735-406-2 (cited on page 8).

[21] Blizzard, *Hearthstone - card library*, `https://playhearthstone.com/en-gb/cards?set=wild`, Accessed on 19.10.2019 (cited on page 101).

[22] P. Bontrager, A. Khalifa, A. Mendes, and J. Togelius, "Matching Games and Algorithms for General Video Game Playing", *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 122–128, 2016 (cited on page 41).

[23] C. Borgelt, M. Steinbrecher, and R. Kruse, *Graphical Models*. Chichester, UK: John Wiley & Sons, Ltd, Aug. 2009, ISBN: 9780470749555 (cited on pages 31, 59).

[24] R. Bouckaert, "Bayesian belief networks: From construction to inference", PhD thesis, University of Utrecht, Utrecht, Netherlands, 1995 (cited on page 60).

[25] B. Bouzy and T. Cazenave, "Computer Go: An AI oriented survey", *Artificial Intelligence*, vol. 132, no. 1, pp. 39–103, Oct. 2001, ISSN: 00043702 (cited on page 2).

[26] S. R. Branavan, D. Silver, and R. Barzilay, "Non-linear Monte-Carlo search in civilization II", *IJCAI International Joint Conference on Artificial Intelligence*, pp. 2404–2410, 2011, ISSN: 10450823 (cited on page 4).

[27] L. Breiman, *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001 (cited on page 85).

[28] G. V. den Broeck, K. Driessens, and J. Ramon, "Monte-carlo tree search in poker using expected reward distributions", in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2009, pp. 367–381 (cited on page 8).

[29] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012, ISSN: 1943-068X (cited on pages 36, 38, 39).

[30] E. Bursztein, "I am a legend: Hacking hearthstone using statistical learning methods", in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, Sep. 2016, pp. 1–8, ISBN: 978-1-5090-1883-3 (cited on pages 104, 107–109).

[31] M. Campbell, A. Hoane, and F.-h. Hsu, "Deep Blue", *Artificial Intelligence*, vol. 134, no. 1-2, pp. 57–83, Jan. 2002, ISSN: 00043702 (cited on pages 2, 24).

[32] T. Cazenave, "A Phantom-Go program", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4250 LNCS, 2006, pp. 120–125, ISBN: 3540488871 (cited on page 8).

[33] A. Champandard and P. Dunstan, "The Behavior Tree Starter Kit", *Game AI Pro: Collected Wisdom of Game AI Professionals*, pp. 73–91, 2013 (cited on page 43).

[34] A. Champandard, "Behavior trees for next-gen game ai", in *Game developers conference, audio lecture*, 2007 (cited on page 43).

[35] A. J. Champandard, "Monte-carlo tree search in TOTAL WAR: ROME IIs campaign ai", *AIGameDev. com: http://aigamedev. com/open/coverage/mcts-rome-ii*, 2014 (cited on page 4).

[36] N. Charness, "Expertise in chess: the balance between knowledge and search", in *Toward a General Theory of Expertise: Prospects and Limits*, K. A. Ericsson and J. Smith, Eds., Cambridge University Press, 1991, pp. 39–64, ISBN: 0-521-40470-3 (cited on page 31).

[37] Y. Chen, H. Yuan, and Y. Li, "Object-Oriented State Abstraction in Reinforcement Learning for Video Games", in *Proceedings of the 2019 IEEE Conference on Games (COG)*, London: IEEE, 2019, pp. 1–4, ISBN: 9781728118840 (cited on page 73).

[38] J. Clune, "Heuristic evaluation functions for general game playing", *Proceedings of the National Conference on Artificial Intelligence*, vol. 2, pp. 1134–1139, 2007 (cited on page 29).

[39] M. Colledanchise, R. Parasuraman, and P. Ogren, "Learning of Behavior Trees for Autonomous Agents", *IEEE Transactions on Games*, vol. 11, no. 2, pp. 183–189, 2018, ISSN: 2475-1502 (cited on page 43).

[40] G. F. Cooper and E. Herskovits, "A Bayesian method for the induction of probabilistic networks from data", *Machine Learning*, vol. 9, no. 4, pp. 309–347, Oct. 1992, ISSN: 0885-6125 (cited on page 60).

[41] P. Cortez and M. J. Embrechts, "Opening black box Data Mining models using Sensitivity Analysis", *IEEE SSCI 2011: Symposium Series on Computational Intelligence - CIDM 2011: 2011 IEEE Symposium on Computational Intelligence and Data Mining*, pp. 341–348, 2011 (cited on page 76).

[42] M. Costalba, J. Kiiski, G. Linscott, and T. Romstad, *Stockfish*, `http://www.stockfishchess.org`, Accessed on 10.12.2019, 2019 (cited on page 36).

[43] "darkfriend77", *Sabberstone github repository*, Accessed on 06.03.2018 (cited on page 136).

[44] G. Díaz and A. Iglesias, "Intelligent Behavioral Design of Non-player Characters in a FPS Video Game Through PSO", in, vol. 1, 2017, pp. 246–254, ISBN: 9783319618241 (cited on page 32).

[45] A. Dockhorn, *Source code and results of forward model learning experiments*, Accessed on 10.12.2019 (cited on page 85).

[46] ——, *Source code and results of predictive state determinization experiments*, Accessed on 10.12.2019 (cited on page 122).

[47] A. Dockhorn and D. Apeldoorn, “Forward Model Approximation for General Video Game Learning”, in *Proceedings of the 2018 IEEE Conference on Computational Intelligence and Games (CIG18)*, IEEE, Aug. 2018, pp. 425–432, ISBN: 9781538643594 (cited on page 80).

[48] A. Dockhorn, C. Braune, and R. Kruse, “An Alternating Optimization Approach based on Hierarchical Adaptations of DBSCAN”, in *2015 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2015, pp. 749–755, ISBN: 9781479975600 (cited on page 126).

[49] A. Dockhorn, C. Doell, M. Hewelt, and R. Kruse, “A decision heuristic for Monte Carlo tree search doppelkopf agents”, in *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, Nov. 2017, pp. 1–8, ISBN: 978-1-5386-2726-6 (cited on pages 4, 31, 52, 121).

[50] A. Dockhorn, M. Frick, U. Akkaya, and R. Kruse, “Predicting Opponent Moves for Improving Hearthstone AI”, in *17th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, IPMU 2018*, J. Medina, M. Ojeda-Aciego, J. L. Verdegay, D. A. Pelta, I. P. Cabrera, B. Bouchon-Meunier, and R. R. Yager, Eds., Springer International Publishing, 2018, pp. 621–632 (cited on page 47).

[51] A. Dockhorn and R. Kruse, “Detecting Sensor Dependencies for Building Complementary Model Ensembles”, in *Proceedings. 28. Workshop Computational Intelligence, Dortmund, 29.-30. November 2018*, 2018, pp. 217–234 (cited on pages 59, 80).

[52] A. Dockhorn, S. M. Lucas, V. Volz, I. Bravi, R. D. Gaina, and D. Perez-Liebana, “Learning Local Forward Models on Unforgiving Games”, in *IEEE Conference on Games (COG)*, London: IEEE, Aug. 2019, pp. 1–4, ISBN: 9781728118840 (cited on pages 81, 93).

[53] A. Dockhorn and S. Mostaghim, “Introducing the Hearthstone-AI Competition”, pp. 1–4, May 2019 (cited on page 136).

[54] A. Dockhorn, C. Saxton, and R. Kruse, “Association Rule Mining for Unknown Video Games”, in *A fuzzy dictionary of fuzzy modelling. Common concepts and perspectives*, Christophe Marsala and Marie-Jeanne Lesot, Ed., 2018 (cited on pages 31, 82).

[55] A. Dockhorn, T. Schwensfeier, and R. Kruse, “Fuzzy Multiset Clustering for Metagame Analysis”, in *Proceedings of the 2019 Conference of the International Fuzzy Systems Association and the European Society for Fuzzy Logic and Technology (EUSFLAT 2019)*, Paris, France: Atlantis Press, 2019, ISBN: 978-94-6252-770-6 (cited on page 110).

[56] A. Dockhorn, T. Tippelt, and R. Kruse, “Model Decomposition for Forward Model Approximation”, in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, Nov. 2018, pp. 1751–1757, ISBN: 978-1-5386-9276-9 (cited on pages 80, 87, 143).

[57] A. Dosovitskiy, I. Labs, V. Koltun, and I. Labs, “Learning To Act By Predicting the Future”, 2017, pp. 1–14 (cited on page 35).

[58] M. Ester, H. P. Kriegel, J. Sander, and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”, *Second International Conference on Knowledge Discovery and Data Mining*, pp. 226–231, 1996, ISSN: 09758887 (cited on page 116).

[59] A. Fern and P. Lewis, “Ensemble Monte-Carlo Planning : An Empirical Study”, pp. 58–65, 2008 (cited on page 41).

[60] M. Fisher, *Introduction to General Game Learning*, `https://graphics.stanford.edu/~mdfisher/GeneralGameLearning.html`, 2014 (cited on page 3).

[61] Ó. Fontenla-Romero, B. Guijarro-Berdiñas, D. Martinez-Rego, B. Pérez-Sánchez, and D. Peteiro-Barral, “Online machine learning”, in *Efficiency and Scalability Methods for Computational Intellect*, IGI Global, pp. 27–54 (cited on page 55).

[62] R. D. Gaina, J. Liu, S. M. Lucas, and D. Perez-Liebana, “Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing”, in *Lecture Notes in Computer Science (including subseries*

Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10199 LNCS, 2017, pp. 418–434, ISBN: 9783319558486 (cited on pages 5, 40).

[63] R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, “Rolling horizon evolution enhancements in general video game playing”, in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, Aug. 2017, pp. 88–95, ISBN: 978-1-5386-3233-8 (cited on pages 5, 40).

[64] ——, “Project Thyia: A Forever Gameplayer”, in *2019 IEEE Conference on Games (CoG)*, 2019, ISBN: 9781728118840 (cited on page 154).

[65] J. A. Gámez, J. L. Mateo, and J. M. Puerta, “Learning Bayesian networks by hill climbing: efficient methods based on progressive restriction of the neighborhood”, *Data Mining and Knowledge Discovery*, vol. 22, no. 1-2, pp. 106–148, Jan. 2011, ISSN: 1384-5810 (cited on page 60).

[66] P. Garcia-Sanchez, A. Tonda, A. M. Mora, G. Squillero, and J. J. Merelo, “Towards automatic StarCraft strategy generation using genetic programming”, *2015 IEEE Conference on Computational Intelligence and Games, CIG 2015 - Proceedings*, pp. 284–291, 2015 (cited on page 32).

[67] P. García-Sánchez, A. Tonda, A. J. Fernández-Leiva, and C. Cotta, “Optimizing hearthstone agents using an evolutionary algorithm”, *Knowledge-Based Systems*, p. 105 032, 2019, ISSN: 0950-7051 (cited on page 138).

[68] M. Genesereth, N. Love, and B. Pell, “General Game Playing: Overview of the AAAI Competition”, *AI Magazin*, vol. 26, no. 2, pp. 62–72, 2005, ISSN: 0738-4602 (cited on pages 3, 25).

[69] M. Ghavamzadeh, S. Mannor, J. Pineau, and A. Tamar, *Bayesian Reinforcement Learning: A Survey*. 2016, vol. abs/1609.04436. arXiv: 1609.04436 (cited on page 19).

[70] B. Goertzel, “Artificial General Intelligence: Concept, State of the Art, and Future Prospects”, *Journal of Artificial General Intelligence*, vol. 5, no. 1, pp. 1–48, Dec. 2014, ISSN: 1946-0163 (cited on page 1).

[71] B. Goertzel, C. Pennachin, and N. Geisweiller, *Engineering General Intelligence, Part 1*, ser. Atlantis Thinking Machines 1. Paris: Atlantis Press, 2014, vol. 5, pp. 21–32, ISBN: 978-94-6239-026-3 (cited on page 1).

[72] ——, *Engineering General Intelligence, Part 2*, ser. Atlantis Thinking Machines. Paris: Atlantis Press, 2014, vol. 6, ISBN: 978-94-6239-029-4 (cited on page 1).

[73] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, p. 1, ISBN: 9781491925614 (cited on page 31).

[74] D. Ha and J. Schmidhuber, "Recurrent world models facilitate policy evolution", in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Curran Associates, Inc., 2018, pp. 2450–2462 (cited on page 45).

[75] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, "Dream to Control: Learning Behaviors by Latent Imagination", pp. 1–19, 2019 (cited on page 45).

[76] T. Hailesilassie, "Rule Extraction Algorithm for Deep Neural Networks: A Review", vol. 14, no. 7, pp. 376–381, 2016 (cited on page 76).

[77] J. Handl, J. Knowles, and D. B. Kell, "Computational cluster validation in post-genomic data analysis", *Bioinformatics*, vol. 21, no. 15, pp. 3201–3212, 2005, ISSN: 13674803 (cited on page 115).

[78] T. Hastie, S. Rosset, J. Zhu, and H. Zou, "Multi-class AdaBoost", *Statistics and Its Interface*, vol. 2, no. 3, pp. 349–360, 2009, ISSN: 19387989 (cited on pages 85, 193).

[79] J. Heinrich and D. Silver, "Deep reinforcement learning from self-play in imperfect-information games", *CoRR*, vol. abs/1603.01121, 2016. arXiv: `1603.01121` (cited on page 46).

[80] M. Henaff, W. F. Whitney, and Y. LeCun, "Model-Based Planning with Discrete and Continuous Actions", 2017 (cited on page 45).

[81] H. J. Holz and M. H. Loew, “Relative feature importance: A classifier-independent approach to feature selection”, in *Pattern Recognition in Practice IV - Multiple Paradigms, Comparative Studies and Hybrid Systems*, Elsevier, 1994, pp. 473–487 (cited on page 76).

[82] A. K. Hoover, J. Togelius, S. Lee, and F. de Mesentier Silva, “The Many AI Challenges of Hearthstone”, *KI - Künstliche Intelligenz*, Sep. 2019, ISSN: 0933-1875 (cited on page 103).

[83] R. A. Howard, “Dynamic programming and markov processes.”, 1960 (cited on page 33).

[84] E. Ilhan and A. S. Etaner-Uyar, “Monte Carlo tree search with temporal-difference learning for general video game playing”, in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, New York: IEEE, Aug. 2017, pp. 317–324, ISBN: 978-1-5386-3233-8 (cited on pages 5, 41).

[85] D. Isla, “Managing complexity in the halo 2 AI”, in *Game Developer’s Conference*, 2005 (cited on page 43).

[86] K. Jithesh, P. B. Anto, P. K. Reshma, and M. Aravindhan, “Role of Particle Swarm Optimization in Computer Games”, in *Proceedings of Fourth International Conference on Soft Computing for Problem Solving*, K. N. Das, K. Deep, M. Pant, J. C. Bansal, and A. Nagar, Eds., ser. Advances in Intelligent Systems and Computing, vol. 336, New Delhi: Springer India, 2015, pp. 521–531, ISBN: 978-81-322-2219-4 (cited on page 32).

[87] T. Joppen, M. U. Moneke, N. Schroder, C. Wirth, and J. Furnkranz, “Informed hybrid game tree search for general video game playing”, *IEEE Transactions on Games*, vol. 10, no. 1, pp. 78–90, Mar. 2018 (cited on page 30).

[88] N. Justesen and S. Risi, “Learning macromanagement in starcraft from replays using deep learning”, *2017 IEEE Conference on Computational Intelligence and Games, CIG 2017*, pp. 162–169, 2017 (cited on page 31).

[89] A. Katona, R. Spick, V. Hodge, S. Demediuk, F. Block, A. Drachen, and J. A. Walker, "Time to Die: Death Prediction in Dota 2 using Deep Learning", in *2019 IEEE Conference on Games (CoG)*, 2019, ISBN: 9781728118840 (cited on page 31).

[90] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning", in *ECML'06 Proceedings of the 17th European conference on Machine Learning*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293, ISBN: 978-3-540-45375-8 (cited on pages 36, 38).

[91] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", *Advances In Neural Information Processing Systems*, pp. 1–9, 2012, ISSN: 10495258 (cited on page 73).

[92] R. Kruse, C. Borgelt, F. Klawonn, C. Moewes, M. Steinbrecher, and P. Held, "Computational intelligence", 2013 (cited on pages 31, 32, 59).

[93] S. Kullback and R. A. Leibler, "On information and sufficiency", *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951 (cited on page 52).

[94] G. N. Lance and W. T. Williams, "A Generalized Sorting Strategy for Computer Classifications", *Nature*, vol. 212, no. 5058, pp. 218–218, Oct. 1966, ISSN: 0028-0836 (cited on page 116).

[95] M. Levandowsky and D. Winter, "Distance between sets", *Nature*, vol. 234, no. 5323, pp. 34–35, Nov. 1971 (cited on page 114).

[96] J. Levine, C. Congdon, M. Ebner, and G. Kendall, "General video game playing", *Dagstuhl Follow-Ups*, pp. 1–7, 2013, ISSN: 1868-8977 (cited on page 3).

[97] S. Liu, S. J. Louis, and C. Ballinger, "Evolving effective micro behaviors in RTS game", in *2014 IEEE Conference on Computational Intelligence and Games*, IEEE, 2014, ISBN: 9781479935468 (cited on page 32).

[98] S. M. Lucas, A. Dockhorn, V. Volz, C. Bamford, R. D. Gaina, I. Bravi, D. Perez-Liebana, S. Mostaghim, and R. Kruse, “A Local Approach to Forward Model Learning: Results on the Game of Life Game”, in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–8 (cited on page 81).

[99] S. M. Lucas, S. Samothrakis, and D. Pérez, “Fast Evolutionary Adaptation for Monte Carlo Tree Search”, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8602, 2014, pp. 349–360, ISBN: 9783662455227 (cited on page 40).

[100] J. B. MacQueen, “Some Methods for classification and Analysis of Multivariate Observations”, *5th Berkeley Symposium on Mathematical Statistics and Probability 1967*, vol. 1, no. 233, pp. 281–297, 1967, ISSN: 00970433 (cited on page 115).

[101] O.-a. Maillard, R. Munos, and D. Ryabko, “Selecting the State-Representation in Reinforcement Learning”, pp. 1–9, Feb. 2013 (cited on page 34).

[102] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008, ISBN: 0521865719, 9780521865715 (cited on page 116).

[103] D. Margaritis and S. Thrun, “Bayesian network induction via local neighborhoods”, *Advances in Neural Information Processing Systems*, pp. 505–511, 2000, ISSN: 10495258 (cited on page 60).

[104] D. Margaritis, S. Thrun, C. Faloutsos, A. W. Moore, and G. F. Cooper, “Learning Bayesian Network Model Structure from Data”, PhD thesis, Carnegie Mellon University, 2003 (cited on page 60).

[105] H. Markram, K. Meier, T. Lippert, S. Grillner, R. Frackowiak, S. Dehaene, A. Knoll, H. Sompolinsky, K. Verstreken, J. DeFelipe, S. Grant, J. P. Changeux, and A. Sariam, “Introducing the Human Brain Project”, *Procedia Computer Science*, vol. 7, pp. 39–42, 2011, ISSN: 18770509 (cited on page 1).

[106] G. Martínez-Arellano, R. Cant, and D. Woods, “Creating AI characters for fighting games using genetic programming”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 4, pp. 423–434, 2017, ISSN: 1943068X (cited on page 32).

[107] M. Maschler, E. Solan, and S. Zamir, *Game Theory*. Cambridge: Cambridge University Press, 2013, ISBN: 9780511794216 (cited on page 36).

[108] L. L. McQuitty, “Elementary Linkage Analysis for Isolating Orthogonal and Oblique Types and Typal Relevancies”, *Educational and Psychological Measurement*, vol. 17, no. 2, pp. 207–229, Jul. 1957, ISSN: 0013-1644 (cited on page 116).

[109] A. Mendes, J. Togelius, and A. Nealen, “Hyper-heuristic general video game playing”, *IEEE Conference on Computatonal Intelligence and Games, CIG*, 2017, ISSN: 23254289 (cited on page 30).

[110] C. E. Metz, “Basic principles of ROC analysis”, *Seminars in Nuclear Medicine*, vol. 8, no. 4, pp. 283–298, 1978, ISSN: 00012998 (cited on page 52).

[111] I. Millington, *AI for Games*, 3rd editio. CRC Press, 2019, ISBN: 978-1-138-48397-2 (cited on pages 29, 33).

[112] T. M. Mitchell, *Machine Learning*. 1997, ISBN: 0071154671 (cited on page 85).

[113] S. Miyamoto, “Fuzzy multisets and their generalizations”, in *Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View [Workshop on Multiset Processing, WMP 2000, Curtea de Arges, Romania, August 21-25, 2000]*, C. Calude, G. Puaun, G. Rozenberg, and A. Salomaa, Eds., ser. Lecture Notes in Computer Science, vol. 2235, Springer, 2000, pp. 225–236, ISBN: 3-540-43063-6 (cited on pages 110, 112).

[114] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning”, *CoRR*, pp. 1–9, Dec. 2013 (cited on pages 6, 34).

[115] V. Mnih, K. Kavukcuoglu, D. Silver, A. a. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning”, *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 0028-0836 (cited on pages 4, 6, 34).

[116] A. M. Mora-García and J. J. Merelo-Guervós, “Evolving Bots AI in Unreal”, in *Algorithmic and Architectural Gaming Design*, IGI Global, 2012, pp. 134–157, ISBN: 9781466616349 (cited on page 32).

[117] M. Müller, “Computer Go”, *Artificial Intelligence*, vol. 134, no. 1-2, pp. 145–179, Jan. 2002, ISSN: 00043702 (cited on pages 31, 36).

[118] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, “Procedural level generation with answer set programming for general video game playing”, *2015 7th Computer Science and Electronic Engineering Conference, CEEC 2015 - Conference Proceedings*, pp. 207–212, 2015 (cited on page 32).

[119] ——, “HTN fighter: Planning in a highly-dynamic game”, *2017 9th Computer Science and Electronic Engineering Conference, CEEC 2017 - Proceedings*, pp. 189–194, 2017 (cited on page 43).

[120] ——, “A Hybrid Planning and Execution Approach Through HTN and MCTS”, in *Proceedings of the 3rd Workshop on Integrated Planning, Acting, and Execution*, M. Roberts, T. Vaquero, T. Niemueller, and S. Fratini, Eds., Berkeley, 2019, pp. 37–45 (cited on page 43).

[121] ——, “Evolving Game State Evaluation Functions for a Hybrid Planning Approach”, in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–8 (cited on page 32).

[122] X. Neufeld, S. Mostaghim, D. L. Sancho-Pradel, and S. Brand, “Building a Planner: A Survey of Planning Systems Used in Commercial Video Games”, *IEEE Transactions on Games*, vol. 11, no. 2, pp. 91–108, 2017, ISSN: 2475-1502 (cited on page 43).

[123] M. Nicolau, D. Perez-Liebana, M. Oneill, and A. Brabazon, “Evolutionary Behavior Tree Approaches for Navigating Platform Games”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 3, pp. 227–238, 2017, ISSN: 1943068X (cited on page 43).

[124] C. P. Nota and D. J. LaPlante, “Improvements to mcts simulation policies in go”, 2014 (cited on page 39).

[125] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev, “The building blocks of interpretability”, *Distill*, 2018, https://distill.pub/2018/building-blocks (cited on page 76).

[126] M. J. Osborne and D. Tunley, *A course in Game Theory*, First Edit. Oxford University Press, 1994, ISBN: 978-0262650403 (cited on page 7).

[127] J. Pearl, “Asymptotic properties of minimax trees and game-searching procedures”, *Artificial Intelligence*, vol. 14, no. 2, pp. 113–138, 1980, ISSN: 00043702 (cited on page 36).

[128] C. Pelling and H. Gardner, “Two Human-Like Imitation-Learning Bots with Probabilistic Behaviors”, in *Proceedings of the 2019 IEEE Conference on Games (COG)*, London: IEEE, 2019, pp. 1–7, ISBN: 9781728118840 (cited on page 31).

[129] D. Perez, S. Mostaghim, S. Samothrakis, and S. M. Lucas, “Multiobjective Monte Carlo Tree Search for Real-Time Games”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4, pp. 347–360, 2015, ISSN: 1943068X (cited on page 94).

[130] D. Perez, S. Samothrakis, and S. Lucas, “Knowledge-based fast evolutionary MCTS for general video game playing”, in *2014 IEEE Conference on Computational Intelligence and Games*, 2014, ISBN: 9781479935468 (cited on page 40).

[131] D. Perez Liebana, J. Dieskau, M. Hunermund, S. Mostaghim, and S. Lucas, “Open Loop Search for General Video Game Playing”, in *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO ’15*, New York, New York, USA: ACM Press, 2015, pp. 337–344, ISBN: 9781450334723 (cited on pages 5, 6, 39).

[132] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, “General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms”, Feb. 2018 (cited on page 25).

[133] D. Perez-Liebana, S. M. Lucas, R. D. Gaina, J. Togelius, A. Khalifa, and J. Liu, *General Video Game Artificial Intelligence*, 2. Morgan & Claypool Publishers, 2019, vol. 3, pp. 1–191, `https://gaigresearch.github.io/gvgaibook/` (cited on pages 49, 50, 89).

[134] D. Perez-Liebana, S. Mostaghim, and S. M. Lucas, “Multi-objective tree search approaches for general video game playing”, *2016 IEEE Congress on Evolutionary Computation, CEC 2016*, pp. 624–631, 2016 (cited on page 94).

[135] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas, “Analyzing the robustness of general video game playing agents”, in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, Sep. 2016, pp. 1–8, ISBN: 978-1-5090-1883-3 (cited on page 5).

[136] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas, “General Video Game AI: Competition, Challenges and Opportunities”, in *Proceedings of the 30th Conference on Artificial Intelligence (AAAI 2016)*, 2016, pp. 4335–4337, ISBN: 9781577357605 (cited on pages 5, 6, 30, 90).

[137] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couetoux, J. Lee, C.-U. Lim, and T. Thompson, “The 2014 General Video Game Playing Competition”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 229–243, Sep. 2016, ISSN: 1943-068X (cited on page 5).

[138] H. Peters, *Game theory: A Multi-leveled approach*, 2nd Edition. Springer, 2015, ISBN: 9783662469491 (cited on page 36).

[139] J. Pettit and D. Helmbold, “Evolutionary learning of policies for MCTS simulations”, *Foundations of Digital Games 2012, FDG 2012 - Conference Program*, no. c, pp. 212–219, 2012 (cited on page 40).

[140] I. P. Pinto and L. R. Coutinho, “Hierarchical Reinforcement Learning with Monte Carlo Tree Search in Computer Fighting Game”, *IEEE Transactions on Games*, vol. 11, no. 3, pp. 290–295, 2019, ISSN: 2475-1502 (cited on page 41).

[141] M. Ponsen, G. Gerritsen, and G. M.J.-B. Chaslot, “Integrating Opponent Models with Monte-Carlo Tree Search in Poker”, in *Proceedings of the 3rd AAAI Conference on Interactive Decision Theory and Game Theory*, ser. AAAIWS’10-03, AAAI Press, 2010, pp. 37–42, ISBN: 9781577354697 (cited on page 8).

[142] M. Preuss, D. Kozakowski, J. Hagelback, and H. Trautmann, “Reactive strategy choice in StarCraft by means of fuzzy control”, in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, IEEE, Aug. 2013 (cited on page 42).

[143] J. R. Quinlan, *C4.5: Programs For Machine Learning*, 1st, M. B. Morgan, Ed. Morgan Kaufmann Publishers, 1993, ISBN: 9780080500584 (cited on pages 31, 85).

[144] D. R. Reddy *et al.*, “Speech understanding systems: A summary of results of the five-year research effort”, *Department of Computer Science. Camegie-Mell University, Pittsburgh, PA*, vol. 17, 1977 (cited on page 138).

[145] J. Redmon and A. Farhadi, “YOLO9000: Better, faster, stronger”, *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. January, pp. 6517–6525, 2017 (cited on page 73).

[146] A. Rosenberg and J. Hirschberg, “V-measure: A conditional entropy-based external cluster evaluation measure”, *Computational Linguistics*, vol. 1, pp. 410–420, 2007 (cited on page 125).

[147] J. Rubin and I. Watson, “Computer poker: A review”, *Artificial Intelligence*, vol. 175, no. 5-6, pp. 958–987, 2011, ISSN: 00043702 (cited on page 3).

[148] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo Method*. John Wiley & Sons, Inc., Nov. 2016 (cited on page 33).

[149] S. Salzberg, "Distance metrics for instance-based learning", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 542 LNAI P, pp. 399–408, 1991, ISSN: 16113349 (cited on page 66).

[150] B. S. Santos and H. S. Bernardino, "Game State Evaluation Heuristics in General Video Game Playing", *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, vol. 2018-Novem, pp. 147–156, 2019, ISSN: 21596662 (cited on page 30).

[151] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved", *Science*, vol. 317, no. 5844, pp. 1518–1522, 2007, ISSN: 00368075 (cited on page 24).

[152] T. Schaul, "A video game description language for model-based or interactive learning", in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, IEEE, Aug. 2013, pp. 1–8, ISBN: 978-1-4673-5311-3 (cited on page 25).

[153] B. Schwab, *Ai postmortem: Hearthstone*, `https://www.gdcvault.com/play/1019998/AI-Postmortem`, 2014 (cited on page 103).

[154] G. Schwarz, "Estimating the Dimension of a Model", *The Annals of Statistics*, vol. 6, no. 2, pp. 461–464, Mar. 1978, ISSN: 0090-5364 (cited on page 59).

[155] M. Scutari, "Learning bayesian networks with the bnlearn R package", *Journal of Statistical Software*, vol. 35, no. 3, pp. 1–22, 2010 (cited on page 61).

[156] M. Scutari and J.-B. Denis, *Bayesian Networks with Examples in R*. Boca Raton: Chapman and Hall, 2014, ISBN 978-1-4822-2558-7, 978-1-4822-2560-0 (cited on pages 59, 61).

[157] S. Sievers and M. Helmert, "A doppelkopf player based on UCT", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9324, 2015, pp. 151–165, ISBN: 9783319244884 (cited on pages 4, 8, 121).

[158] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search", *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, ISSN: 0028-0836 (cited on pages 2, 24, 40).

[159] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", *Annals of Clinical and Translational Neurology*, vol. 5, no. 1, pp. 92–97, Dec. 2017, ISSN: 23289503 (cited on pages 2, 3, 41).

[160] ——, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play", *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018, ISSN: 10959203 (cited on pages 2, 3, 7, 24, 41).

[161] C. F. Sironi, J. Liu, D. Perez-Liebana, R. D. Gaina, I. Bravi, S. M. Lucas, and M. H. Winands, "Self-adaptive MCTS for General Video Game Playing", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10784 LNCS, pp. 358–375, 2018, ISSN: 16113349 (cited on page 40).

[162] C. F. Sironi and M. H. Winands, "Analysis of Self-Adaptive Monte Carlo Tree Search in General Video Game Playing", *IEEE Conference on Computatonal Intelligence and Games, CIG*, vol. 2018-Augus, pp. 18–21, 2018, ISSN: 23254289 (cited on page 40).

[163] N. Sturtevant, "An analysis of UCT in multi-player games", *ICGA Journal*, vol. 31, no. 4, pp. 195–208, 2008, ISSN: 13896911 (cited on pages 38, 52).

[164] SuperData Research, MMOs.com, *Leading digital collectible card game (ccg) titles worldwide in 2016, by revenue (in million u.s. dollars)*, Statista: `https://www.statista.com/statistics/666594/digital-collectible-card-games-by-revenue/`, 2017 (cited on page 100).

[165] R. S. Sutton, "Learning to predict by the methods of temporal differences", *Machine Learning*, vol. 3, no. 1, pp. 9–44, Aug. 1988, ISSN: 0885-6125 (cited on page 33).

[166] R. S. Sutton and A. G. Barto, *Reinforcement Learning*, 2nd ed. Cambridge: The MIT Press, 2018, ISBN: 9780262039246 (cited on pages 15, 19, 20, 35, 36).

[167] M. Swiechowski, T. Tajmajer, and A. Janusz, "Improving Hearthstone AI by Combining MCTS and Supervised Learning Algorithms", in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, Aug. 2018, pp. 1–8, ISBN: 978-1-5386-4359-4 (cited on page 4).

[168] T. G. Tan, P. Anthony, J. Teo, and J. H. Ong, "Neural network ensembles for video game AI using evolutionary multi-objective optimization", *Proceedings of the 2011 11th International Conference on Hybrid Intelligent Systems, HIS 2011*, pp. 605–610, 2011 (cited on page 41).

[169] E. Tomai and R. Flores, "Adapting In-Game Agent Behavior by Observation of Players Using Learning Behavior Trees", in *Proceedings of the 9th International Conference on the Foundations of Digital Games*, 2014 (cited on page 43).

[170] C. K. Tong, C. Kim On, J. Teo, and A. M.O. J. Kiring, "Evolving Neural controllers using GA for Warcraft 3-Real Time Strategy game", *Proceedings - 2011 6th International Conference on Bio-Inspired Computing: Theories and Applications, BIC-TA 2011*, pp. 15–20, 2011 (cited on page 32).

[171] R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana, "Deep Reinforcement Learning for General Video Game AI", *IEEE Conference on Computatonal Intelligence and Games, CIG*, vol. 2018-Augus, 2018, ISSN: 23254289 (cited on page 35).

[172] J. Tromp and G. Farnebäck, *Combinatorics of Go*, 2016 (cited on page 36).

[173] I. Tsamardinos, C. F. Aliferis, and A. Statnikov, "Time and sample efficient discovery of Markov blankets and direct causal relations",

in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, New York, New York, USA: ACM Press, 2003, p. 673, ISBN: 1581137370 (cited on page 60).

[174] I. Tsamardinos, L. E. Brown, and C. F. Aliferis, "The max-min hill-climbing Bayesian network structure learning algorithm", *Machine Learning*, vol. 65, no. 1, pp. 31–78, 2006, ISSN: 0885-6125 (cited on page 61).

[175] N. Tziortziotis, G. Papagiannis, and K. Blekas, "A Bayesian Ensemble Regression Framework on the Angry Birds Game", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 2, pp. 104–115, 2016, ISSN: 1943068X (cited on page 41).

[176] C. J. Van Rooyen, W. S. Van Heerden, and C. W. Cleghorn, "Playing the game of snake with limited knowledge: Unsupervised neuro-controllers trained using particle swarm optimization", *IEEE 4th International Conference on Soft Computing and Machine Intelligence, ISCMI 2017*, vol. January, pp. 80–84, 2018 (cited on page 32).

[177] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3720 LNAI, pp. 437–448, 2005, ISSN: 03029743 (cited on page 19).

[178] J. C. Villanueva, *How many atoms are there in the universe?*, `https://www.universetoday.com/36302/atoms-in-the-universe/`, 2009 (cited on page 106).

[179] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features", in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1, IEEE Comput. Soc, 2017, pp. 511–518, ISBN: 0-7695-1272-0 (cited on page 73).

[180] Q. H. Vu, D. Ruta, A. Ruta, and L. Cen, "Predicting win-rates of hearthstone decks: Models and features that won AAIA'2018 data mining challenge", *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems, FedCSIS 2018*, vol. 15, pp. 197–200, 2018 (cited on page 31).

[181] M. de Waard, D. M. Roijers, and S. C. Bakkes, “Monte Carlo Tree Search with options for general video game playing”, in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, Sep. 2016, pp. 1–8, ISBN: 978-1-5090-1883-3 (cited on page 5).

[182] C. J.C. H. Watkins and P. Dayan, “Q-learning”, *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992, ISSN: 0885-6125 (cited on page 33).

[183] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Silver, and D. Wierstra, “Imagination-Augmented Agents for Deep Reinforcement Learning”, 2017 (cited on page 45).

[184] D. H. Wolpert and W. G. Macready, “No free lunch theorems for optimization”, *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997, ISSN: 1089778X (cited on page 42).

[185] R. Xu, D. Wunsch, and D. Wunsch II, “Survey of Clustering Algorithms”, *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005, ISSN: 10459227 (cited on page 115).

[186] R. R. Yager, “On the Theory of Bags”, *International Journal of General Systems*, vol. 13, no. 1, pp. 23–37, Nov. 1986, ISSN: 0308-1079 (cited on pages 110–112).

[187] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*. Cham: Springer International Publishing, 2018, ISBN: 978-3-319-63518-7 (cited on pages 17, 23, 34, 42).

[188] A. Yilmaz, O. Javed, and M. Shah, “Object tracking: A survey”, *ACM Computing Surveys*, vol. 38, no. 4, 2006, ISSN: 03600300 (cited on page 73).

[189] C. Zhang and Y. Ma, Eds., *Ensemble Machine Learning*. Boston, MA: Springer US, 2012, ISBN: 978-1-4419-9325-0 (cited on page 41).

[190] Y. Zhang, S. Wang, and G. Ji, “A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications”, *Mathematical Problems in Engineering*, vol. 2015, pp. 1–38, 2015, ISSN: 15635147 (cited on page 32).

[191] D. D. Zhu, D. Whitehouse, E. J. Powley, and P. I. Cowling, “Determinization and information set Monte Carlo Tree Search for the Card Game Dou Di Zhu”, in *2011 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2011 (cited on pages 8, 47, 99).

[192] G. L. Zuin, Y. P. Macedo, L. Chaimowicz, and G. L. Pappa, “Discovering combos in fighting games with evolutionary algorithms”, *GECCO 2016 - Proceedings of the 2016 Genetic and Evolutionary Computation Conference*, pp. 277–284, 2016 (cited on page 32).

List of Figures

3.1 Steps of the monte carlo tree search algorithm. 38
3.2 Comparison of general game-playing and -learning techniques based on knowledge of the return function and the game's model. 46

4.1 Screenshot of the game Paku Paku 58
4.2 Two levels of the game Sokoban 61
4.3 Comparison of learned dependence structures 64
4.4 Unit circles of the Minkowski distance for varying values of p . 67
4.5 Visualising steps of the local forward model's prediction process. 68
4.6 Screenshot of the game "aliens" of the GVGAI framework . . . 71
4.7 Visual comparison of object-based forward model architectures 71
4.8 Visual comparison of proposed forward model architectures . 77
4.9 Prediction-based search including continuous model updates. 79
4.10 Three games of the GVGAI framework 83
4.11 Local neighbourhood patterns encoding the local neighbourhood of the centre tile for span sizes 1 and 3 85
4.12 Aggregated accuracy values measured on all evaluation games 86
4.13 Agent comparison by rank per game. 91
4.14 Agent comparison by rank per game. 92
4.15 Hierarchical agglomerative clustering of each game's ranking vector using complete linkage. 93
4.16 Ranking of the agents' performance in a continuous learning setup 96
4.17 Ranking of the agents' performance in a transfer learning setup 98

5.1 Elements of the Hearthstone game board 101
5.2 Examples of general card types 102
5.3 2D-embedding obtained by Multi Dimensional Scaling of the deck space of the Hunter hero class. 105
5.4 3-phase action-selection using predictive models for state determinisation 123

5.5 Distance matrices for the deck dataset 126
5.6 Grid-search results for the optimisation of DBSCAN parameters. 127
5.7 Comparison of clustering results based on external validation measures homogeneity, completeness, and the v-measure . . 128
5.8 Accuracy for predicting cards that may appear in the remaining turns of the game bucketed by turn. 130
5.9 Accuracy for the aggregated prediction of cards that may appear in the remaining turns of the game bucketed by turn. 130
5.10 Accuracy for the prediction of cards that may appear in the remaining turns of the game bucketed by turn. 131
5.11 Accuracy for the aggregated prediction of cards that may appear in the remaining turns of the game bucketed by turn. 131
5.12 Accuracy for predicting cards that may appear in the remaining turns of the game bucketed by turn. 134
5.13 Accuracy for the aggregated prediction of cards that may be played by the opponent during its next turn bucketed by turn. 134
5.14 Accuracy for predicting cards that may appear in the remaining turns of the game bucketed by turn. 135
5.15 Accuracy for the aggregated prediction of cards that may be played by the opponent during its next turn bucketed by turn. 135
5.16 Win-rates for each match-up after 360 simulated games. . . . 140
5.17 Average win-rate per combination of decks. 141
5.18 Win-rate for each match-up when only considering decks the proposed agent was trained for. 142
5.19 Proposed agent's average win-rate per deck while playing against a single opponent. 144
5.19 Proposed agent's average win-rate per deck while playing against a single opponent. 145

A.1 Games of the GVGAI framework 190
A.2 Aggregated accuracy values measured per game 194
A.3 Average performance per epoch in the continuous learning evaluation. 216

List of Tables

4.1 Comparison of structure-learning algorithms for Sokoban . . 63
4.2 Qualitative comparison of proposed forward models architectures 76
4.3 Tracked attributes per object instance 87
4.4 Aggregated ranks over all tested games and final score per agent 90
4.5 Aggregated ranks over all tested games and final score per agent 95
4.6 Aggregated ranks over all tested games and final score per agent 97

5.1 Exemplary results of proposed counting schemes. 108
5.2 General contingency matrix 124
5.3 Best performing parameter configuration per clustering algorithm 127
5.4 Comparison of the methods' average prediction accuracy in discussed evaluation scenarios. 133

A.1 Classifier parameters and grid-search options 193
A.2 Best performing combination of algorithm, parameters, and data sets per game for learning a local forward model. 201
A.3 Constant model performance - bait 202
A.4 Constant model performance - catapults 202
A.5 Constant model performance - chainreaction 202
A.6 Constant model performance - chase 203
A.7 Constant model performance - chipschallenge 203
A.8 Constant model performance - clusters 203
A.9 Constant model performance - colourescape 204
A.10 Constant model performance - decepticoins 204
A.11 Constant model performance - deceptizelda 204
A.12 Constant model performance - doorkoban 205
A.13 Constant model performance - escape 205
A.14 Constant model performance - fireman 205
A.15 Constant model performance - garbagecollector 206
A.16 Constant model performance - hungrybirds 206
A.17 Constant model performance - iceandfire 206

A.18 Constant model performance - islands 207
A.19 Constant model performance - labyrinth 207
A.20 Constant model performance - labyrinthdual 207
A.21 Constant model performance - painter 208
A.22 Constant model performance - realsokoban 208
A.23 Constant model performance - run 208
A.24 Constant model performance - shipwreck 209
A.25 Constant model performance - sokoban 209
A.26 Constant model performance - surround 209
A.27 Constant model performance - tercio 210
A.28 Constant model performance - thecitadel 210
A.29 Constant model performance - thesnowman 210
A.30 Constant model performance - vortex 211
A.31 Constant model performance - watergame 211
A.32 Constant model performance - whackamole 211
A.33 Continuous learning performance - bait 212
A.34 Continuous learning performance - catapults 212
A.35 Continuous learning performance - chipschallenge 212
A.36 Continuous learning performance - clusters 212
A.37 Continuous learning performance - decepticoins 213
A.38 Continuous learning performance - escape 213
A.39 Continuous learning performance - garbagecollector 213
A.40 Continuous learning performance - hungrybirds 213
A.41 Continuous learning performance - iceandfire 214
A.42 Continuous learning performance - islands 214
A.43 Continuous learning performance - labyrinth 214
A.44 Continuous learning performance - labyrinthdual 214
A.45 Continuous learning performance - painter 215
A.46 Continuous learning performance - run 215
A.47 Continuous learning performance - watergame 215
A.48 Transfer learning performance - bait 219
A.49 Transfer learning performance - catapults 219
A.50 Transfer learning performance - chipschallenge 219
A.51 Transfer learning performance - clusters 219
A.52 Transfer learning performance - decepticoins 220

A.53 Transfer learning performance - escape 220
A.54 Transfer learning performance - garbagecollector 220
A.55 Transfer learning performance - hungrybirds 220
A.56 Transfer learning performance - iceandfire 221
A.57 Transfer learning performance - islands 221
A.58 Transfer learning performance - labyrinth 221
A.59 Transfer learning performance - labyrinthdual 221
A.60 Transfer learning performance - painter 222
A.61 Transfer learning performance - run 222
A.62 Transfer learning performance - watergame 222

Appendix

A.1 Games of the GVGAI Framework

bait: The agent controls a small knight, which is trapped in a dungeon. To escape it, he needs to collect the key and get out through the door. Grey boxes that block the path to the key need to be pushed away but pushing them into the key will destroy it. Later levels include holes that can be filled by pushing a block into them and mushrooms that award bonus points upon collection.

catapults: The agent needs to reach the portal without touching any water tiles. Moving on a grey block sends the agent flying in the indicated direction until an object blocks its path. Reaching the final door requires the agent to plan a lot of steps, since landing on the wrong island may trap him.

chainreaction: The agent controls a beast that tries to push boxes into goalposts to score points. Once the red box is pushed into a direction, it will continue until it hits something else. Grey blocks of the same shape can be moved by the agent but do not continue in the same direction. The agent dies in case he walks into a black hole.

chase The agent tries to catch birds, which are always moving directly away from him. Birds can be trapped in corners and dead-ends. When catching a bird, a worm is dropped in the same position. In case any other

bird touches a worm it will turn into a raven. The raven will fly to the agent using the shortest path. As soon as the raven touches the agent the game is lost.

chipschallenge: Each level includes up to four coloured blocks which cannot be passed unless the agent collected a potion of the same colour. The agent can increase its score by collecting coins. A door, which is blocking the path to the final exit, is destroyed in case the agent collected at least 11 coins before touching the door. The game is won in case the agent reaches the final exit and lost in case the agent walks on a fire or water tile. However, dying can be avoided by first collecting boots of the same colour. Chipschallenge is one of the more complex games using larger levels and requiring the agent to manage multiple resources.

clusters: The agent controls a knight which is trying to group blocks of the same colour. Blocks marked with a grey square can be pushed into any direction as long as the next tile is empty. Solid blocks cannot be pushed. The agent wins as soon as all blue and all red blocks form a single group and loses in case he touches any of the spikes.

colourescape: In this game, the agent's goal is to reach the door. While walking around the grid, the player can push grey blocks in any direction unless the target tile is blocked by another object. Touching a blue or red tile changes the agent's colours and allows him to push blocks of the same colour. Touching any of the spikes kills the agent and loses the game.

decepticoins: The agent needs to traverse a maze to reach the target position at the bottom right corner while collecting as many coins as possible. Two paths are available. While taking the left one will provide the agent with a small reward, choosing the longer path will reward more points. Walking past the first corner will spawn a block that blocks the agent's path, such that it is impossible to collect the coins on both paths. The game is won as soon as the agent reaches the block in the bottom right corner.

deceptizelda: In deceptizelda, the agent needs to escape a dungeon through one of the two available doors. While one of the doors is open from the start,

the other one needs to be unlocked using a key. Escaping through the closed door yields more points. However, the key is guarded by multiple enemies which can damage and kill the agent. To kill the enemies, the agent can use the action button to strike its sword in the direction he is currently facing. Killing an enemy awards additional points which makes this a high risk and high reward strategy.

doorkoban: The agent's goal is to reach the final portal. Doors can be opened by moving boxes on switches. However, it is unknown which switch opens which door.

escape: The agent steers a mouse through a maze and tries to reach the cheese. Boxes can be pushed to open new paths. The game does only provide a point when reaching the goal. Most levels require the agent to plan since a single mistake can ultimately block the way to the goal.

fireman: The agent controls a fireman which is tasked to put out burning houses throughout the level. Therefore, it first needs to collect water at a hydrant. Extinguishing a fire yields 2 points and the game is won as soon as all fires have been extinguished. However, getting close to a building that is on fire drains the agent's health and possibly kills him.

garbagecollector: The agent's goal is to collect all black objects that are spread throughout the map. Every time the agent moves to another tile leaves a deadly block on its old position. Collecting the objects in a way that does not hinder the agent to collect remaining objects is key to winning this game.

hungrybirds: The agent controls a bird, which tries to reach the goal in a limited amount of steps. The number of remaining steps can be reset by eating a worm which is hidden within the level. Therefore, the agent needs to decide if it is possible to reach the goal without eating the worm. Exceeding the step limit loses the game.

iceandfire: The agent needs to traverse the forest to reach a doorway. Meanwhile, the agent can collect coins to score points. Fire or ice tiles are deadly until the agent collected the correct shoes. Nevertheless, walking on a spike will always kill the agent.

islands: The agent starts on an island surrounded by water. While standing on sand tiles, it can dig to remove the sand and receive one block of dirt. Walking on a water tile while holding a dirt block turns it into a dirt tile while leaving the agent unharmed. Reaching the gate wins the game. Since the agent cannot swim it is crucial to use dirt blocks efficiently. Otherwise, the agent will get stuck on an island without any more dirt blocks to collect.

labyrinth: The agent represented by the small man in the bottom left corner needs to traverse the labyrinth to reach the goalpost. Walking on a spike will kill the agent.

labyrinthdual: Similar to the game labyrinth with the addition of coloured blocks. In case the agent touches the blue or the red house, the agent's colour changes accordingly allowing him to pass blocks of the same colour. Later levels allow the agent to get stuck in case it changes its colour in an inescapable position.

painter: The game consists of tiles that are either coloured or white. Every time the agent leaves a tile, its state is switched. The agent's goal is to colour all the tiles. When colouring a tile the agent receives one point. Later levels include darkened tiles, which can be passed but do not change the state. These do not count into the winning condition but can enable faster paths to solve the level.

realsokoban: This game mimics the original Sokoban game in which the agent is tasked to push all the boxes on the tiles marked by a circle. However, boxes cannot be pulled. This can hinder the agent from finishing a level since the box may not be movable from any direction.

run: The agent tries to escape the flood by walking to the gate. Water is fastly spreading from its initial position. While trying to escape the agent needs to avoid dead-ends and collect keys to open doors. Points are only rewarded in case the agent reaches the goal.

shipwreck: The agent controls a ship which needs to stay away from tornados. Every few game-steps a collectable spawns in the sea, which can be collected. When bringing these to a harbour the agent receives a reward.

sokoban: Similar to “realsokoban” the agent needs to push boxes onto the target goals. However, when doing so the boxes disappear in this version of sokoban. The game is won as soon as all boxes are removed from the game board.

surround: Here, the agent controls the yeti and plays against one or multiple wolves. Every time the yeti moves, the underlying tile will change the colour to green. Tiles touched by a wolve will be coloured in brown. The agent’s goal is to colour as many tiles as possible before choosing to stop the game. However, the agent loses the game when touching any of the brown tiles or stopping the game too late.

tercio: In this game, the agent needs to push a tree to a hole in the ground. The game consists of the ground tiles water, snow, earth, and dark earth. While the tree can be pushed to any neighbouring tile, the agent can only move in between pre-defined pairs of tiles. Specifically, the agent can walk back and forth between water and earth, earth and dark earth, and earth and snow.

thecitadel: In this game, the agent needs to reach the door to finish the level. Once again the agent is able to push boxes by walking into them. In contrast to previous games, multiple boxes can be pushed as long as the field behind these boxes is empty.

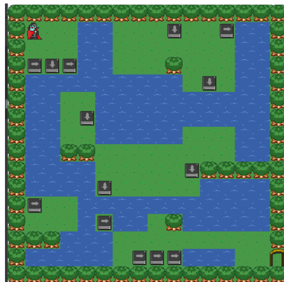
thesnowman: The agent needs to build a snowman by pushing the snowballs of increasing size into each other. While the first level consists of an open field, more advances levels have maze-like structures.

vortex: In this game, the agent needs to reach the door to win the game. Touching water will make the agent lose control and pull him towards the vortex in the middle of the level. New ground tiles can be created by pushing any of the grey blocks into the water. When doing so, the agent receives 1 point. Additionally, the agent receives 10 points when collecting the chest.

watergame: The agent needs to reach the door to escape the dungeon. Water tiles can only be passed in case the agent previously collected any of the blue potions. Walking on a water tile removes the potion from the agent's inventory.

whackamole: From time to time a mole comes out of its molehill for a short time. The agent tries to touch as many of them as possible while avoiding to touch the randomly moving cat. The game is won in case the agent survives 2000 game steps. Every time a mole is touched the agent receives a point. Next to surviving, the agent's performance will be rated by the number of points gained.

a) bait

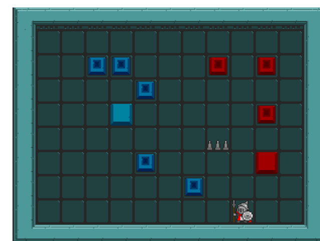


b) catapults

c) chainreation

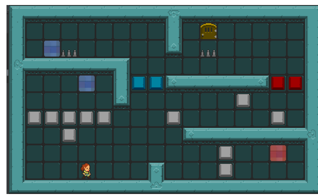
d) chase

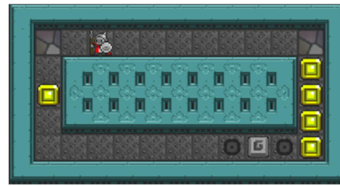
e) chipschallenge



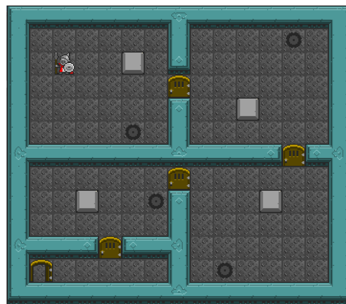
f) clusters

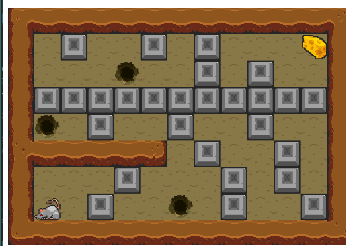
Figure A.1: Games of the GVGAI framework used for the evaluation of forward model learning.

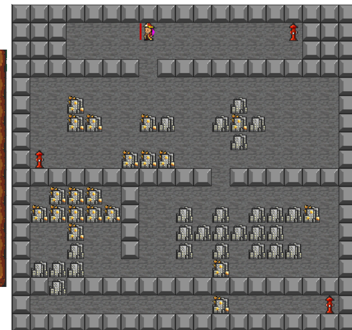

g) colourescape

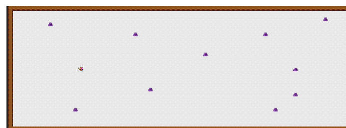

h) decepticoins

i) deceptizelda


j) doorkoban

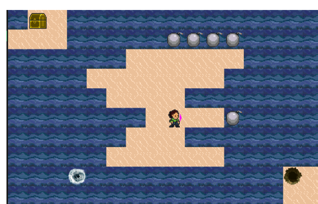

k) escape

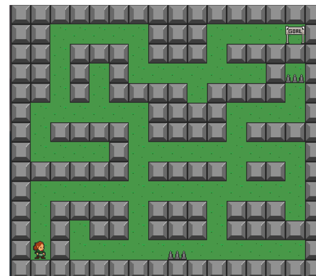

l) fireman

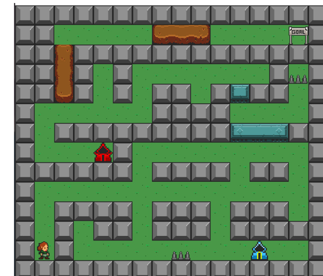

m) garbagecollector

n) hungrybirds

o) iceandfire

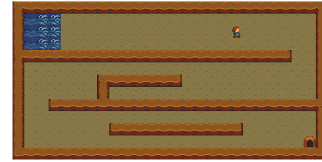

p) islands


q) labyrinth

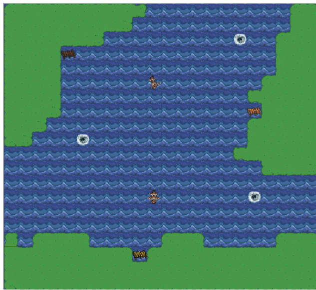

r) labyrinthdual

s) painter

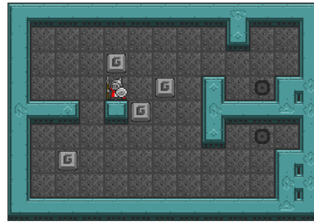
t) realsokoban



u) run



v) shipwreck



w) sokoban

x) surround

y) tercio

z) thecitadel

aa) thesnowman

ab) vortex

ac) watergame

ad) whackamole

A.2 Grid-Search for Tuning Local Forward Models

Table A.1: Classifier parameters and grid-search options

a) k-Nearest Neighbour parameters

Parameter	Parameter Options
number of neighbours	$\{1, 5, 10\}$
metric	minkowski distance

b) Decision Tree parameters

Parameter	Parameter Options
max depth	$\{1, 5, 10\}$
min samples split	2
min samples leaf	1
split criterion	{*random splits*, *best gini gain*}

c) Random Forest parameters

Parameter	Parameter Options
max depth	$\{1, 5, 10\}$
min samples split	2
number of estimators	{5, 10, 100}
split criterion	{*random splits*, *best gini gain*}

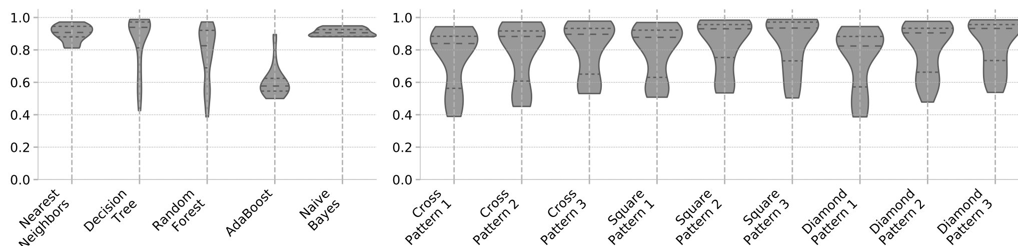
d) AdaBoost parameters

Parameter	Parameter Options
number of estimators	{5, 10, 100}
learning rate	1.0
algorithm	SAMME.R [78]

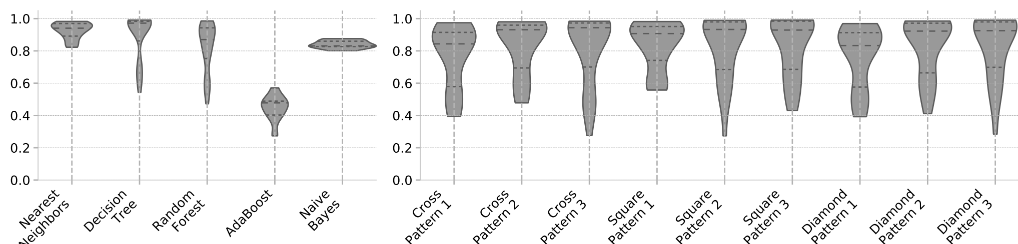
e) Naïve Bayes parameters

Parameter	Parameter Options
no parameters to tune	

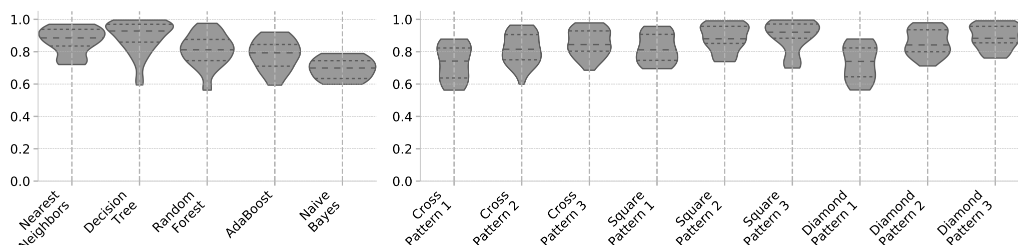
A.3 Local Forward Model Accuracy per Game



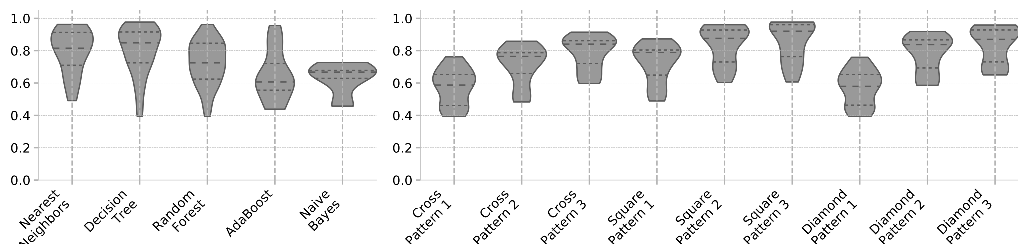
a) local forward model accuracy - “bait”



b) local forward model accuracy - “catapults”

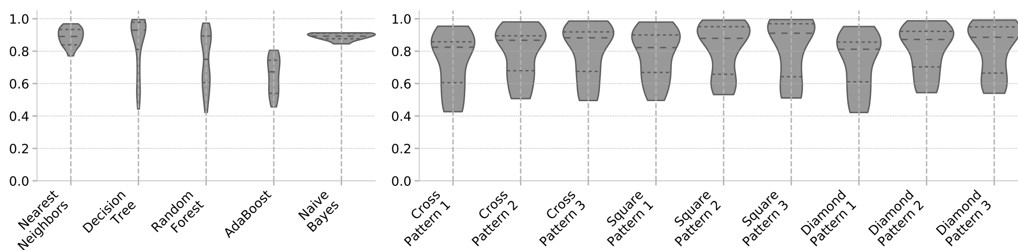


c) local forward model accuracy - “chainreaction”

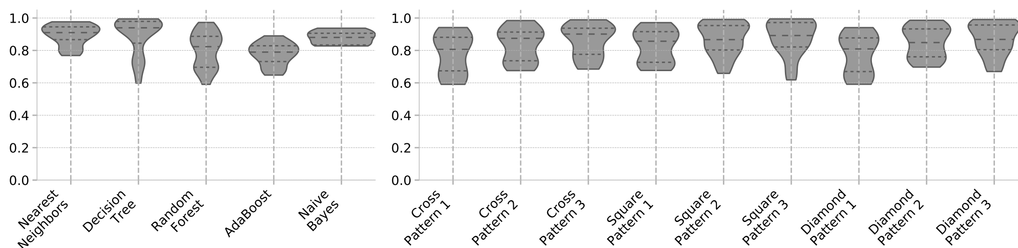


d) local forward model accuracy - “chase”

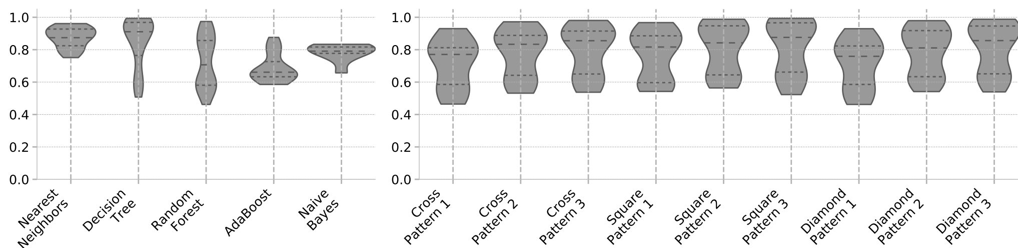
Figure A.2: Aggregated accuracy values measured per game: (left) performance distribution per algorithm (right) performance distribution per data set



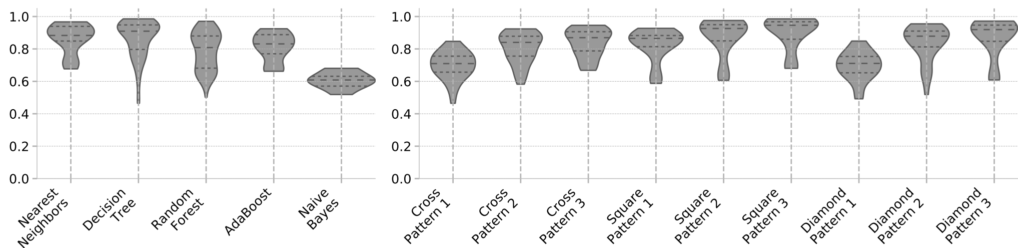
e) local forward model accuracy - “chipschallenge”



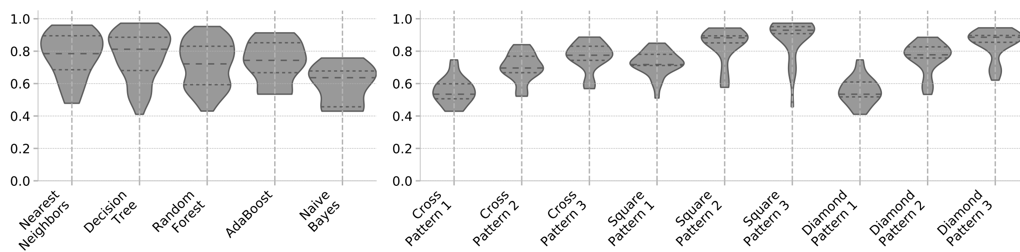
f) local forward model accuracy - “clusters”



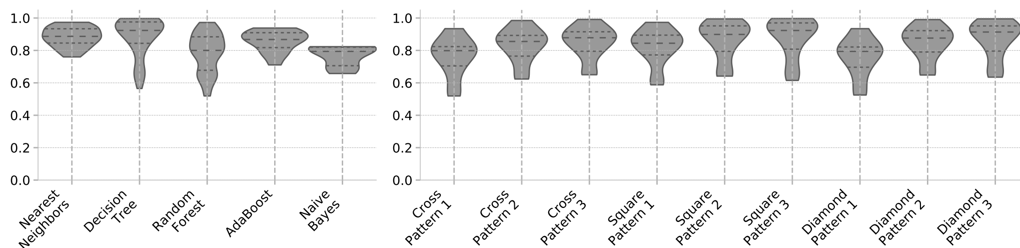
g) local forward model accuracy - “colourescape”



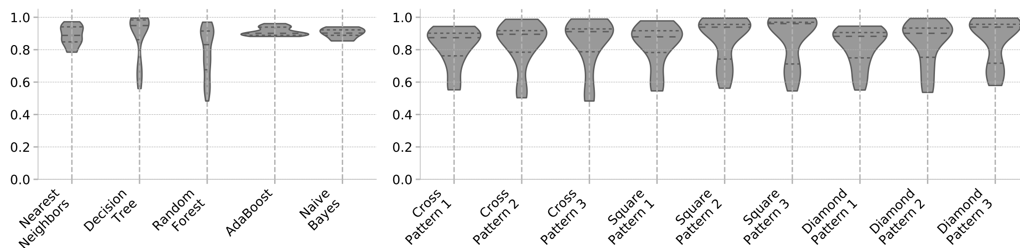
h) local forward model accuracy - “decepticoins”



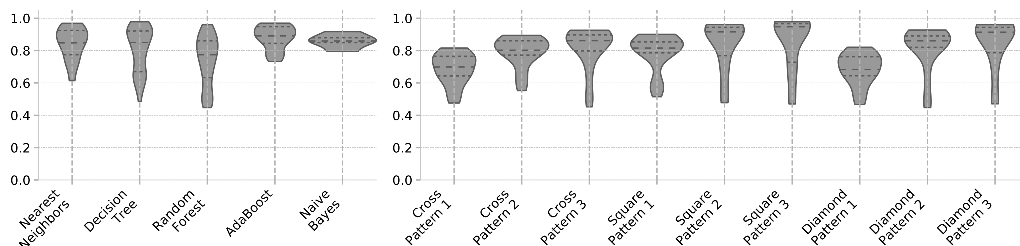
i) local forward model accuracy - "deceptizelda"



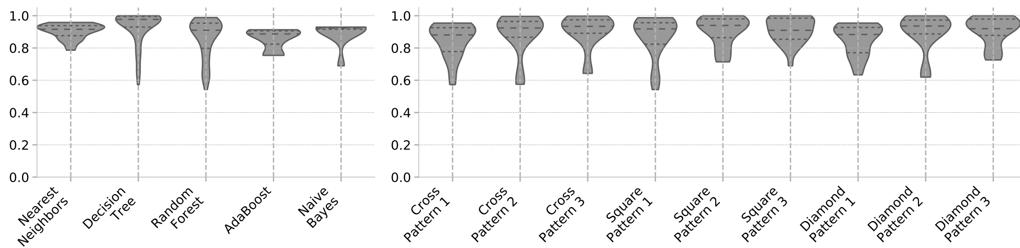
j) local forward model accuracy - "doorkoban"



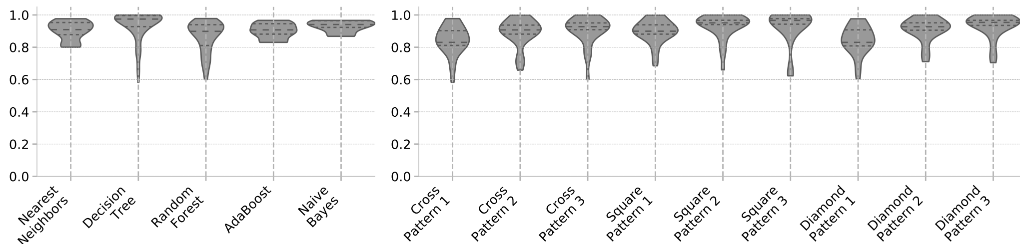
k) local forward model accuracy - "escape"



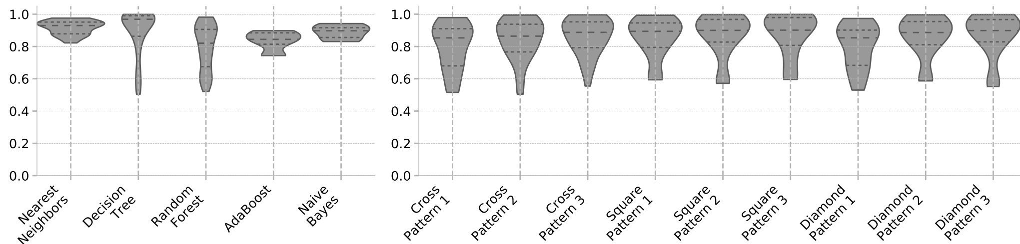
l) local forward model accuracy - "fireman"



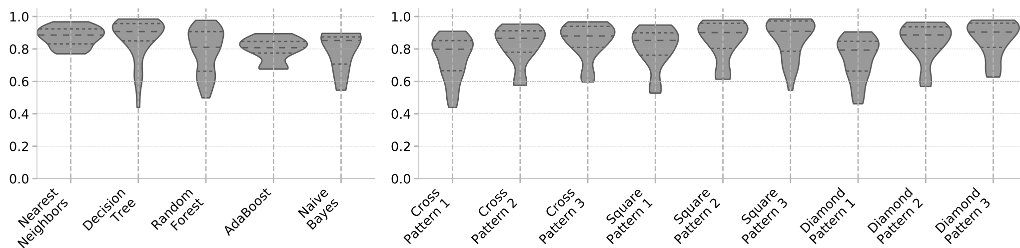
m) local forward model accuracy - “garbagecollector”



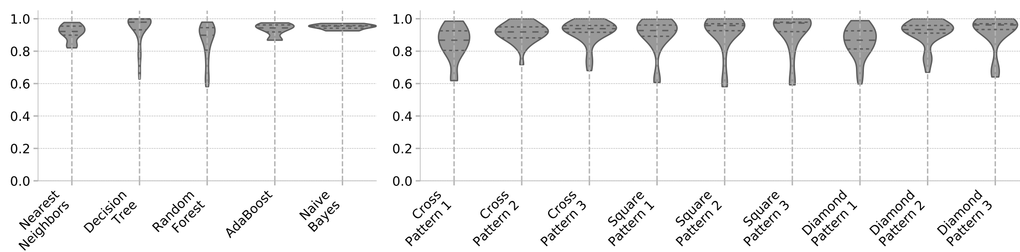
n) local forward model accuracy - “hungrybirds”



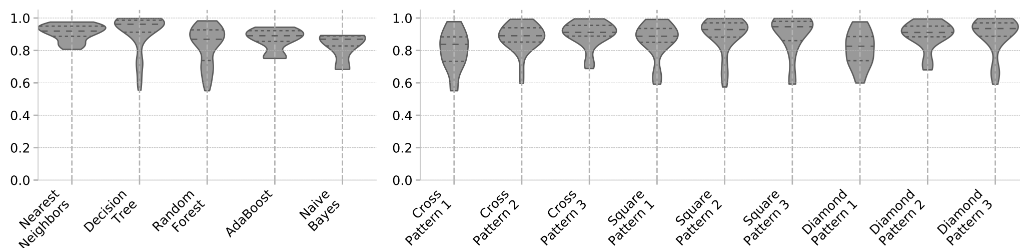
o) local forward model accuracy - “iceandfire”



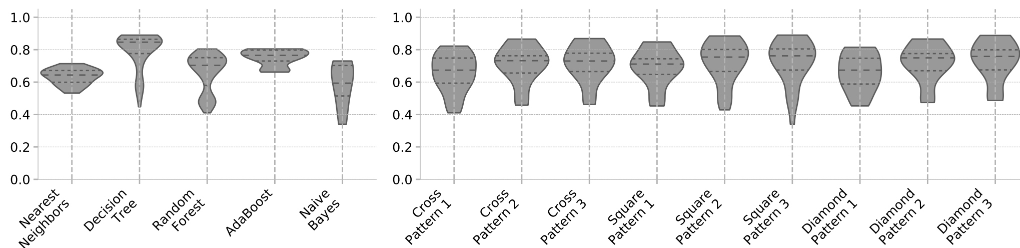
p) local forward model accuracy - “islands”



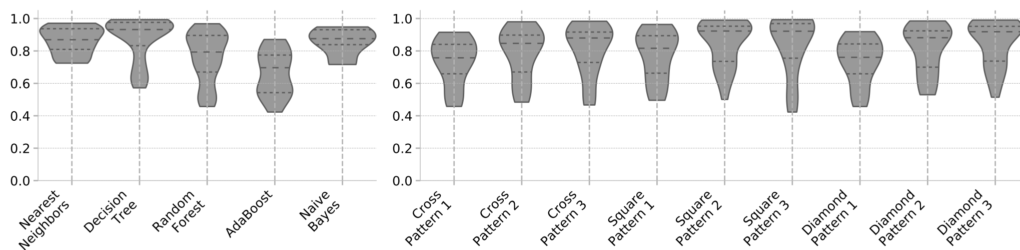
q) local forward model accuracy - “labyrinth”



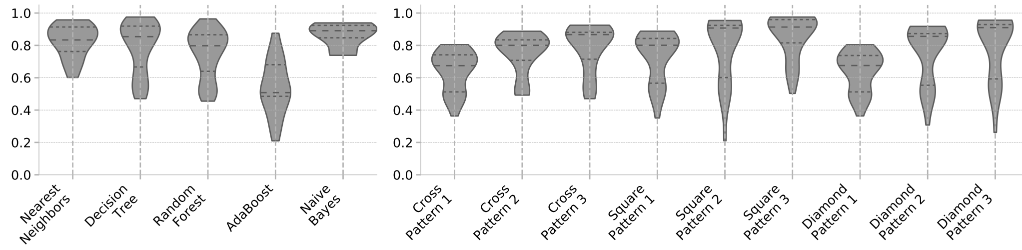
r) local forward model accuracy - “labyrinthdual”



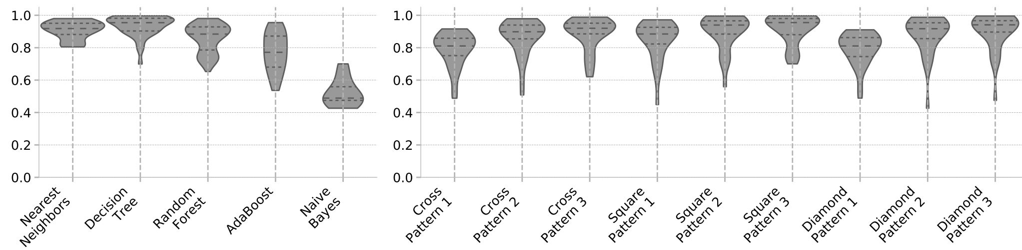
s) local forward model accuracy - “painter”



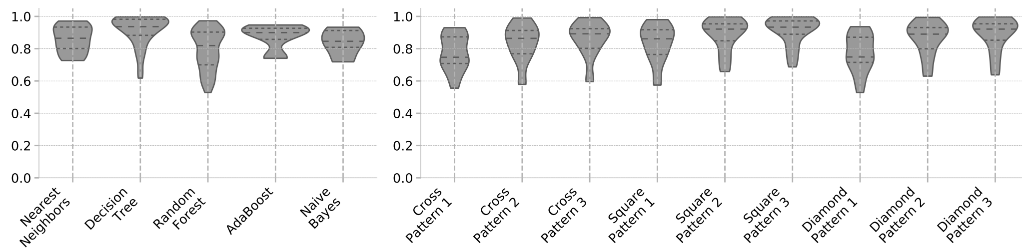
t) local forward model accuracy - “realsokoban”



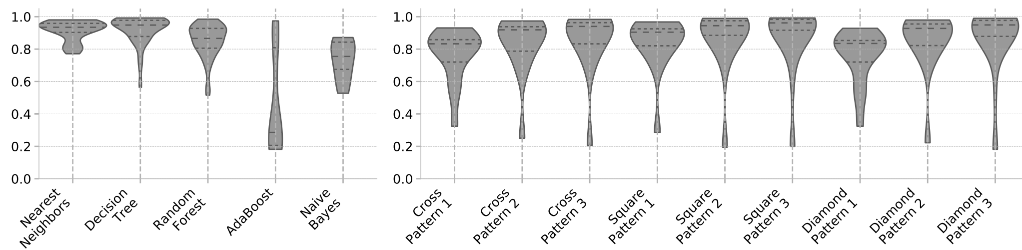
u) local forward model accuracy - “run”



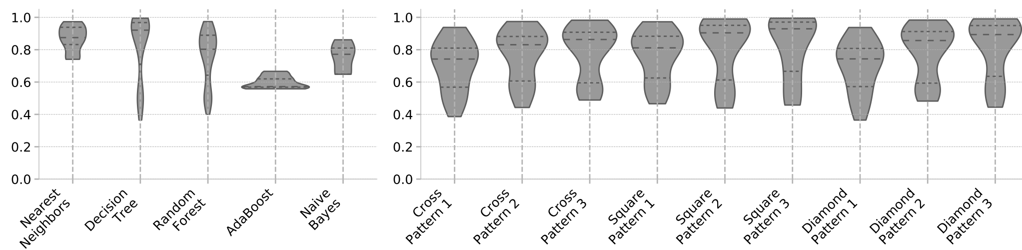
v) local forward model accuracy - “shipwreck”



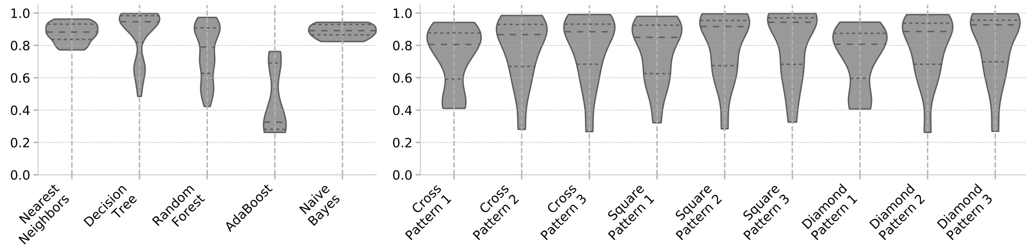
w) local forward model accuracy - “sokoban”



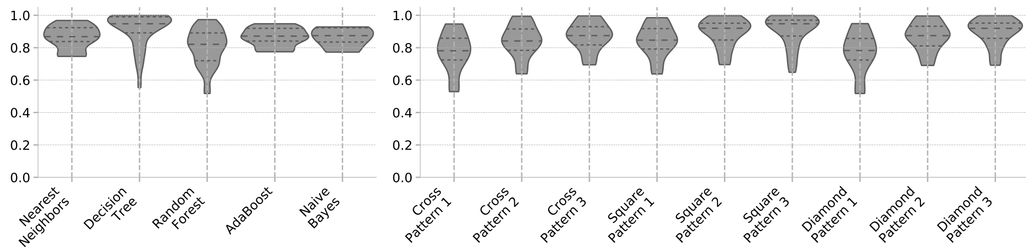
x) local forward model accuracy - “surround”



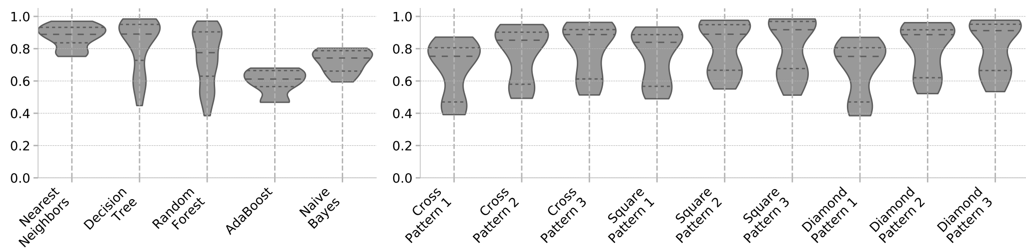
y) local forward model accuracy - “tercio”



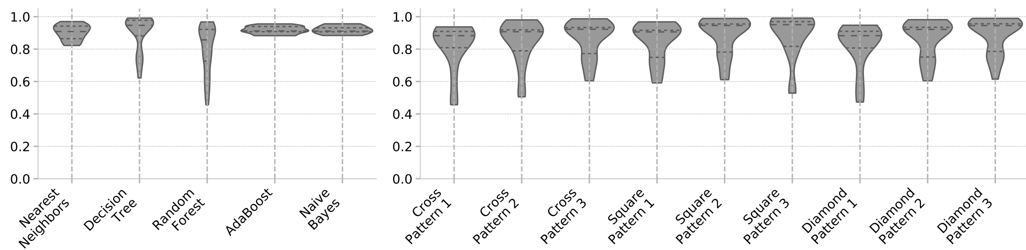
z) local forward model accuracy - “thecitadel”



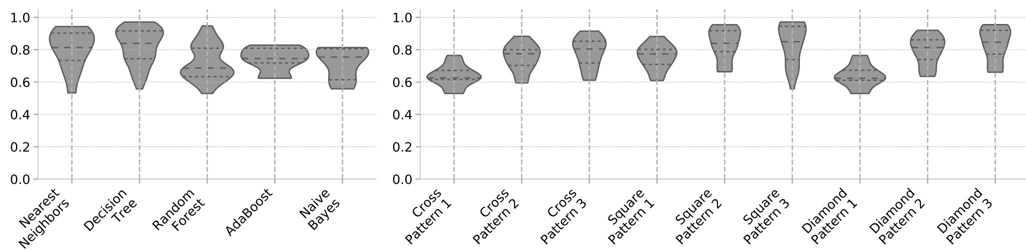
aa) local forward model accuracy - “thesnowman”



ab) local forward model accuracy - “vortex”



ac) local forward model accuracy - “watergame”



ad) local forward model accuracy - “whackamole”

Table A.2: Best performing combination of algorithm, parameters, and data sets per game for learning a local forward model.

Game	Classifier	Parameters		Data Set	Mean
		max depth	split		Accuracy
bait	DT	None	best	Square 3	0.99
catapults	DT	10	best	Square 3	0.99
chainreaction	DT	None	best	Square 3	1.0
chase	DT	10	best	Square 3	0.98
chipschallenge	DT	None	best	Square 3	1.0
clusters	DT	None	best	Square 3	0.99
colourescape	DT	None	best	Square 3	0.99
decepticoins	DT	10	best	Square 3	0.99
deceptizelda	DT	10	random	Square 3	0.97
doorkoban	DT	None	best	Square 3	1.0
escape	DT	None	best	Diamond 3	0.99
fireman	DT	5	best	Square 3	0.98
garbagecollector	DT	10	best	Square 3	1.0
hungrybirds	DT	None	best	Square 3	1.0
iceandfire	DT	None	best	Square 3	1.0
islands	DT	5	best	Square 3	0.98
labyrinth	DT	None	best	Cross 3	1.0
labyrinthdual	DT	None	best	Square 3	1.0
painter	DT	5	random	Square 3	0.89
realsokoban	DT	None	best	Square 3	0.99
run	DT	10	best	Square 3	0.98
shipwreck	DT	None	best	Square 3	1.0
sokoban	DT	None	best	Square 3	1.0
surround	DT	5	best	Square 3	0.99
tercio	DT	None	best	Square 3	0.99
thecitadel	DT	None	best	Square 3	1.0
thesnowman	DT	None	best	Square 3	1.0
vortex	DT	10	best	Square 3	0.98
watergame	DT	None	best	Square 3	0.99
whackamole	DT	10	best	Square 3	0.97

A.4 Constant Model Game-Playing Performance

Table A.3: Constant model performance - bait

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.06	1.28	111.33	427.3	7th
LFM	BFS	0.2	2.18	33.7	561.3	3rd
	RHEA	0.12	1.52	73.33	337.3	6th
	MCTS	0.18	1.98	76.33	317.05	4th
OBFM	BFS	0.2	2.52	9.2	1566.58	1st
	RHEA	0.12	2.12	117.83	656.91	5th
	MCTS	0.2	2.44	15.0	1998.0	2nd

Table A.4: Constant model performance - catapults

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.38	—	10.9	7th
LFM	BFS	0.0	1.12	—	383.66	3rd
	RHEA	0.0	0.76	—	14.26	6th
	MCTS	0.0	1.12	—	15.12	4th
OBFM	BFS	0.0	1.44	—	776.68	1st
	RHEA	0.0	1.2	—	217.08	2nd
	MCTS	0.0	0.76	—	1544.68	5th

Table A.5: Constant model performance - chainreaction

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-9.8	—	308.78	7th
LFM	BFS	0.0	-3.28	—	1114.74	4th
	RHEA	0.0	-7.36	—	770.2	6th
	MCTS	0.0	-3.16	—	1203.52	3rd
OBFM	BFS	0.0	0.0	—	1498.0	1st
	RHEA	0.0	-4.8	—	1075.74	5th
	MCTS	0.0	-2.72	—	1192.02	2nd

Table A.6: Constant model performance - chase

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	0.58	—	1697.04	6th
LFM	BFS	0.0	0.38	—	1745.24	7th
	RHEA	0.0	0.72	—	1837.72	4th
	MCTS	0.0	0.88	—	1896.9	3rd
OBFM	BFS	0.0	1.12	—	991.32	2nd
	RHEA	0.0	0.68	—	1769.32	5th
	MCTS	0.02	0.94	1608.0	1439.49	1st

Table A.7: Constant model performance - chipschallenge

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	4.22	—	418.9	3rd
LFM	BFS	0.0	2.12	—	854.1	6th
	RHEA	0.0	5.36	—	422.74	2nd
	MCTS	0.0	1.64	—	501.1	7th
OBFM	BFS	0.0	2.44	—	1736.66	4th
	RHEA	0.0	6.26	—	1121.48	1st
	MCTS	0.0	2.2	—	1998.0	5th

Table A.8: Constant model performance - clusters

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	-0.86	—	58.15	7th
LFM	BFS	0.0	0.2	—	1175.72	2nd
	RHEA	0.0	-0.24	—	357.42	6th
	MCTS	0.0	0.18	—	1198.38	3rd
OBFM	BFS	0.0	0.14	—	567.04	4th
	RHEA	0.0	0.12	—	389.12	5th
	MCTS	0.0	0.4	—	1468.14	1st

Table A.9: Constant model performance - colourescape

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-0.7	—	604.56	7th
LFM	BFS	0.0	-0.06	—	1419.42	2nd
	RHEA	0.06	-0.58	896.0	789.72	1st
	MCTS	0.0	-0.16	—	1385.38	3rd
OBFM	BFS	0.0	-0.38	—	1057.02	5th
	RHEA	0.0	-0.64	—	845.14	6th
	MCTS	0.0	-0.16	—	1314.48	4th

Table A.10: Constant model performance - decepticoins

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.54	1.8	667.78	326.65	4th
LFM	BFS	0.66	2.8	23.15	684.82	1st
	RHEA	0.58	2.68	607.55	327.81	2nd
	MCTS	0.54	3.28	366.81	507.52	3rd
OBFM	BFS	0.2	1.64	52.1	1634.4	7th
	RHEA	0.38	2.2	90.79	1349.0	6th
	MCTS	0.4	1.58	43.0	1630.33	5th

Table A.11: Constant model performance - deceptizelda

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.52	1.08	745.15	1666.42	4th
LFM	BFS	0.76	1.72	583.16	1458.83	1st
	RHEA	0.48	0.88	483.5	1243.27	6th
	MCTS	0.58	1.44	722.55	1692.71	2nd
OBFM	BFS	0.0	-0.66	—	1082.46	7th
	RHEA	0.54	1.16	755.59	1418.39	3rd
	MCTS	0.5	1.14	579.52	1527.88	5th

Table A.12: Constant model performance - doorkoban

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.24	—	998.0	6th
LFM	BFS	0.0	0.04	—	998.0	7th
	RHEA	0.0	0.46	—	998.0	4th
	MCTS	0.0	0.48	—	998.0	3rd
OBFM	BFS	0.0	0.7	—	998.0	1st
	RHEA	0.0	0.36	—	998.0	5th
	MCTS	0.0	0.5	—	998.0	2nd

Table A.13: Constant model performance - escape

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-1.0	—	117.96	7th
LFM	BFS	0.04	-0.44	199.5	533.29	3rd
	RHEA	0.1	-0.72	302.6	370.62	1st
	MCTS	0.0	-0.34	—	762.92	6th
OBFM	BFS	0.0	-0.26	—	864.2	5th
	RHEA	0.06	-0.76	515.33	425.49	2nd
	MCTS	0.0	0.0	—	998.0	4th

Table A.14: Constant model performance - fireman

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-13.9	—	977.6	5th
LFM	BFS	0.0	-12.54	—	859.3	2nd
	RHEA	0.0	-13.88	—	973.32	4th
	MCTS	0.0	-14.64	—	954.6	7th
OBFM	BFS	0.0	-14.34	—	1141.78	6th
	RHEA	0.0	-10.04	—	568.24	1st
	MCTS	0.0	-12.94	—	894.26	3rd

Table A.15: Constant model performance - garbagecollector

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-1.0	—	14.72	7th
LFM	BFS	0.0	0.6	—	42.16	2nd
	RHEA	0.0	-0.44	—	32.58	4th
	MCTS	0.0	1.2	—	86.42	1st
OBFM	BFS	0.0	-0.44	—	24.08	5th
	RHEA	0.0	-0.48	—	30.6	6th
	MCTS	0.0	0.16	—	52.52	3rd

Table A.16: Constant model performance - hungrybirds

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.8	—	356.5	5th
LFM	BFS	0.96	96.0	65.79	523.0	1st
	RHEA	0.0	0.8	—	329.5	6th
	MCTS	0.02	2.0	292.0	325.55	4th
OBFM	BFS	0.42	42.0	166.9	367.83	2nd
	RHEA	0.02	3.6	228.0	357.69	3rd
	MCTS	0.0	0.0	—	343.0	7th

Table A.17: Constant model performance - iceandfire

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	1.5	—	230.78	7th
LFM	BFS	0.0	5.72	—	204.52	1st
	RHEA	0.0	3.38	—	236.04	4th
	MCTS	0.0	4.6	—	295.82	3rd
OBFM	BFS	0.0	4.92	—	370.68	2nd
	RHEA	0.0	3.34	—	359.62	5th
	MCTS	0.0	3.26	—	443.56	6th

Table A.18: Constant model performance - islands

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.0	—	23.1	7th
LFM	BFS	0.0	0.0	—	84.48	3rd
	RHEA	0.0	0.0	—	35.12	6th
	MCTS	0.0	2.0	—	36.04	1st
OBFM	BFS	0.0	0.0	—	85.38	2nd
	RHEA	0.0	0.0	—	63.1	4th
	MCTS	0.0	0.0	—	49.6	5th

Table A.19: Constant model performance - labyrinth

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.02	-0.62	462.0	565.31	7th
LFM	BFS	0.3	-0.26	362.33	341.17	1st
	RHEA	0.04	-0.2	483.5	875.44	4th
	MCTS	0.08	-0.18	572.5	878.11	3rd
OBFM	BFS	0.2	-0.5	277.6	399.3	2nd
	RHEA	0.04	-0.38	927.5	775.44	5th
	MCTS	0.02	-0.32	933.0	854.27	6th

Table A.20: Constant model performance - labyrinthdual

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.32	—	364.16	5th
LFM	BFS	0.0	2.46	—	998.0	1st
	RHEA	0.0	1.22	—	715.9	3rd
	MCTS	0.0	2.38	—	952.84	2nd
OBFM	BFS	0.0	0.68	—	495.12	4th
	RHEA	0.0	0.2	—	681.38	6th
	MCTS	0.0	0.02	—	827.7	7th

Table A.21: Constant model performance - painter

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.22	447.72	245.64	1998.0	4th
LFM	BFS	0.96	146.98	222.29	1998.0	2nd
	RHEA	0.48	423.8	397.17	1998.0	3rd
	MCTS	1.0	150.54	305.08	—	1st
OBFM	BFS	0.0	7.34	—	1998.0	7th
	RHEA	0.22	420.16	471.64	1998.0	5th
	MCTS	0.0	51.48	—	1998.0	6th

Table A.22: Constant model performance - realsokoban

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.06	—	1998.0	7th
LFM	BFS	0.0	0.64	—	1998.0	4th
	RHEA	0.0	0.1	—	1998.0	6th
	MCTS	0.0	0.54	—	1998.0	5th
OBFM	BFS	0.0	1.06	—	1998.0	2nd
	RHEA	0.0	1.0	—	1998.0	3rd
	MCTS	0.0	1.16	—	1998.0	1st

Table A.23: Constant model performance - run

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.0	—	72.8	6th
LFM	BFS	0.0	0.0	—	75.04	5th
	RHEA	0.0	0.0	—	76.02	4th
	MCTS	0.0	0.0	—	71.06	7th
OBFM	BFS	0.04	0.04	379.0	128.62	1st
	RHEA	0.0	0.0	—	94.18	3rd
	MCTS	0.02	0.02	294.0	142.63	2nd

Table A.24: Constant model performance - shipwreck

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-9.7	—	139.7	7th
LFM	BFS	0.96	6.98	998.0	261.0	4th
	RHEA	0.56	2.02	998.0	658.0	6th
	MCTS	1.0	8.04	998.0	—	1st
OBFM	BFS	0.98	0.22	998.0	25.0	3rd
	RHEA	0.76	2.76	998.0	388.83	5th
	MCTS	1.0	4.56	998.0	—	2nd

Table A.25: Constant model performance - sokoban

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.04	0.54	515.5	1998.0	6th
LFM	BFS	0.0	0.3	—	1998.0	7th
	RHEA	0.08	0.7	260.25	1998.0	5th
	MCTS	0.1	0.68	631.8	1998.0	4th
OBFM	BFS	0.2	0.6	7.0	1998.0	2nd
	RHEA	0.12	0.76	402.67	1998.0	3rd
	MCTS	0.22	0.86	205.82	1998.0	1st

Table A.26: Constant model performance - surround

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		1.0	1.06	6.68	—	1st
LFM	BFS	1.0	0.3	1.14	—	4th
	RHEA	1.0	0.66	2.88	—	3rd
	MCTS	1.0	0.24	1.44	—	5th
OBFM	BFS	0.92	4.22	11.91	757.25	7th
	RHEA	1.0	0.86	4.88	—	2nd
	MCTS	0.96	4.92	18.77	24.0	6th

Table A.27: Constant model performance - tercio

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.0	—	998.0	1st
LFM	BFS	0.0	0.0	—	998.0	1st
	RHEA	0.0	0.0	—	998.0	1st
	MCTS	0.0	0.0	—	998.0	1st
OBFM	BFS	0.0	0.0	—	998.0	1st
	RHEA	0.0	0.0	—	998.0	1st
	MCTS	0.0	0.0	—	998.0	1st

Table A.28: Constant model performance - thecitadel

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.1	1.12	675.8	1998.0	1st
LFM	BFS	0.0	0.2	—	1998.0	6th
	RHEA	0.04	0.72	596.0	1998.0	4th
	MCTS	0.0	0.16	—	1998.0	7th
OBFM	BFS	0.0	0.4	—	1998.0	5th
	RHEA	0.08	1.14	357.0	1998.0	2nd
	MCTS	0.06	0.74	452.0	1998.0	3rd

Table A.29: Constant model performance - thesnowman

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.54	—	1878.72	1st
LFM	BFS	0.0	0.48	—	1697.46	5th
	RHEA	0.0	0.44	—	1776.9	6th
	MCTS	0.0	0.52	—	1893.82	2nd
OBFM	BFS	0.0	0.32	—	1546.0	7th
	RHEA	0.0	0.5	—	1890.14	3rd
	MCTS	0.0	0.5	—	1854.3	4th

Table A.30: Constant model performance - vortex

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.14	—	998.0	7th
LFM	BFS	0.0	0.24	—	998.0	1st
	RHEA	0.0	0.2	—	998.0	2nd
	MCTS	0.0	0.16	—	998.0	6th
OBFM	BFS	0.0	0.2	—	998.0	2nd
	RHEA	0.0	0.18	—	998.0	5th
	MCTS	0.0	0.2	—	998.0	2nd

Table A.31: Constant model performance - watergame

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.0	—	38.62	7th
LFM	BFS	0.06	0.0	43.67	661.43	1st
	RHEA	0.0	0.0	—	66.58	5th
	MCTS	0.0	0.0	—	51.28	6th
OBFM	BFS	0.0	0.0	—	1382.64	3rd
	RHEA	0.0	0.0	—	620.62	4th
	MCTS	0.0	0.0	—	1998.0	2nd

Table A.32: Constant model performance - whackamole

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	1.76	—	201.37	7th
LFM	BFS	0.3	6.3	498.0	145.57	1st
	RHEA	0.22	3.5	498.0	125.72	5th
	MCTS	0.2	5.7	498.0	131.72	6th
OBFM	BFS	0.24	1.68	498.0	108.89	4th
	RHEA	0.28	6.06	498.0	125.94	2nd
	MCTS	0.28	5.42	498.0	204.69	3rd

A.5 Continuous Learning Game-Playing Performance

Table A.33: Continuous learning performance - bait

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.06	1.28	111.33	427.3	2nd
LFM	BFS	0.16	2.64	19.88	149.69	1st
OBFM	BFS	0.0	1.0	—	716.22	3rd

Table A.34: Continuous learning performance - catapults

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	0.38	—	10.9	3rd
LFM	BFS	0.0	0.74	—	9.91	2nd
OBFM	BFS	0.0	1.02	—	178.75	1st

Table A.35: Continuous learning performance - chipschallenge

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	4.22	—	418.9	1st
LFM	BFS	0.0	2.27	—	436.54	3rd
OBFM	BFS	0.0	2.86	—	1060.39	2nd

Table A.36: Continuous learning performance - clusters

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	-0.86	—	58.15	3rd
LFM	BFS	0.0	-0.52	—	120.5	2nd
OBFM	BFS	0.0	-0.42	—	100.8	1st

Table A.37: Continuous learning performance - decepticoins

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.54	1.8	667.78	326.65	2nd
LFM	BFS	0.6	4.76	72.23	76.05	1st
OBFM	BFS	0.0	1.98	—	1225.92	3rd

Table A.38: Continuous learning performance - escape

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-1.0	—	117.96	3rd
LFM	BFS	0.04	-0.92	31.0	107.4	1st
OBFM	BFS	0.0	-0.22	—	861.56	2nd

Table A.39: Continuous learning performance - garbagecollector

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-1.0	—	14.72	3rd
LFM	BFS	0.0	-1.0	—	54.42	2nd
OBFM	BFS	0.0	-0.96	—	15.98	1st

Table A.40: Continuous learning performance - hungrybirds

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.8	—	356.5	3rd
LFM	BFS	0.94	94.0	76.43	448.0	1st
OBFM	BFS	0.0	76.4	—	515.38	2nd

Table A.41: Continuous learning performance - iceandfire

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	1.5	—	230.78	3rd
LFM	BFS	0.06	6.3	208.0	70.04	1st
OBFM	BFS	0.0	6.33	—	307.06	2nd

Table A.42: Continuous learning performance - islands

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	0.0	—	23.1	3rd
LFM	BFS	0.0	0.0	—	133.0	2nd
OBFM	BFS	0.0	0.0	—	245.56	1st

Table A.43: Continuous learning performance - labyrinth

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.02	-0.62	462.0	565.31	2nd
LFM	BFS	0.72	0.44	101.31	100.64	1st
OBFM	BFS	0.0	0.72	—	110.46	3rd

Table A.44: Continuous learning performance - labyrinthdual

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	0.32	—	364.16	3rd
LFM	BFS	0.28	4.32	138.21	377.72	1st
OBFM	BFS	0.0	2.36	—	480.3	2nd

Table A.45: Continuous learning performance - painter

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.22	447.72	245.64	1998.0	2nd
LFM	BFS	1.0	80.1	156.14	—	1st
OBFM	BFS	0.0	23.08	—	1998.0	3rd

Table A.46: Continuous learning performance - run

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.0	—	72.8	3rd
LFM	BFS	0.0	0.0	—	92.78	1st
OBFM	BFS	0.0	0.0	—	90.16	2nd

Table A.47: Continuous learning performance - watergame

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.0	—	38.62	2nd
LFM	BFS	0.0	0.0	—	25.26	3rd
OBFM	BFS	0.0	0.0	—	1041.7	1st

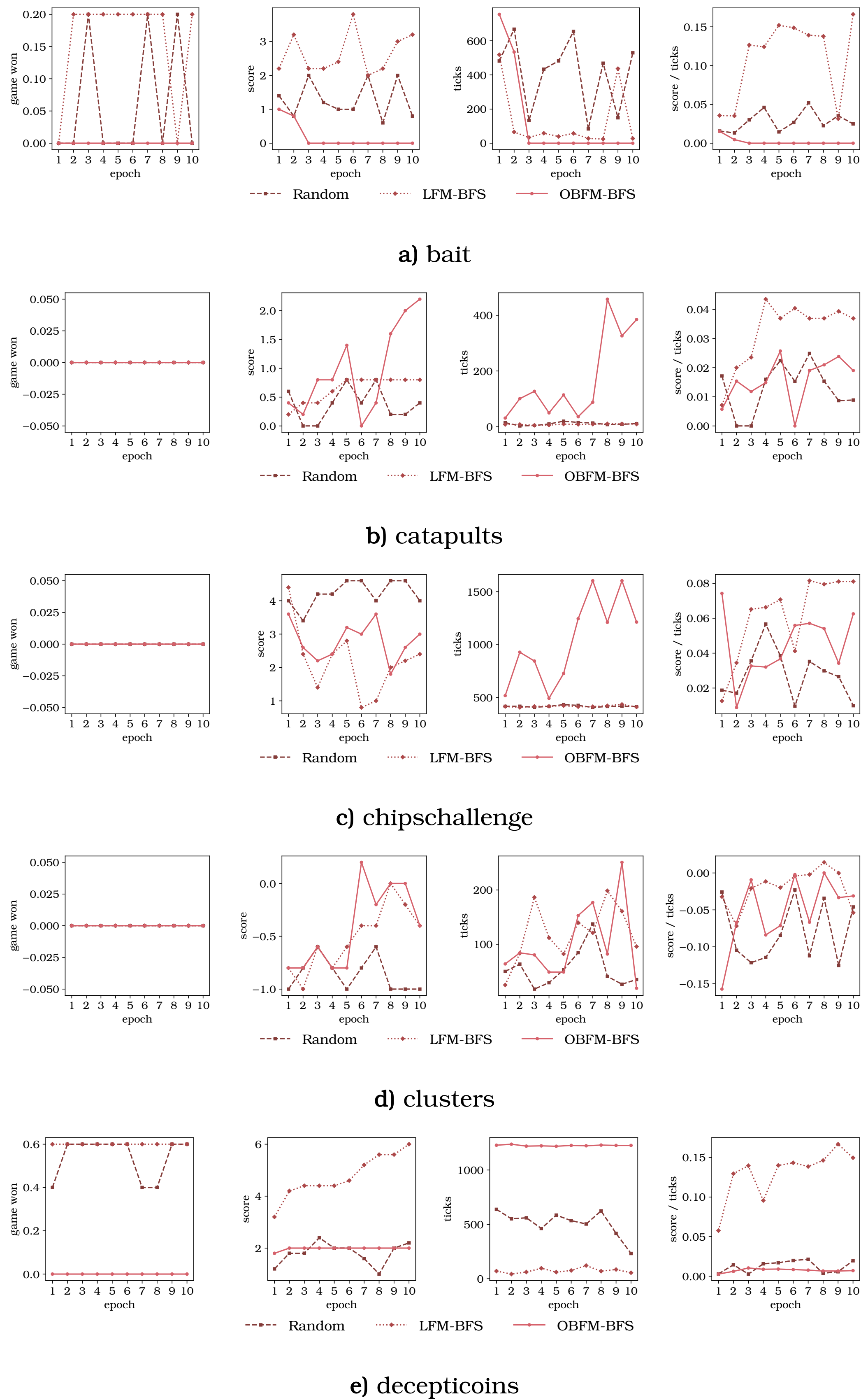
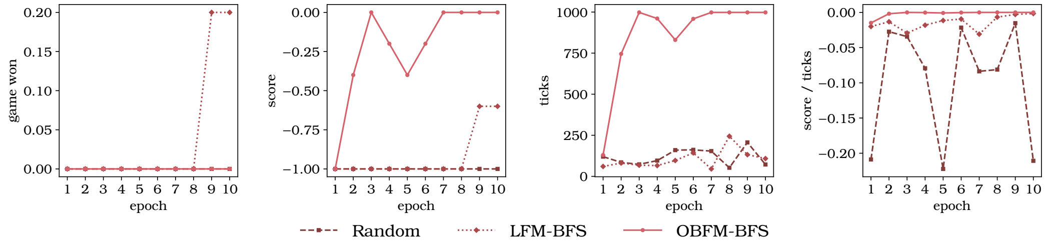
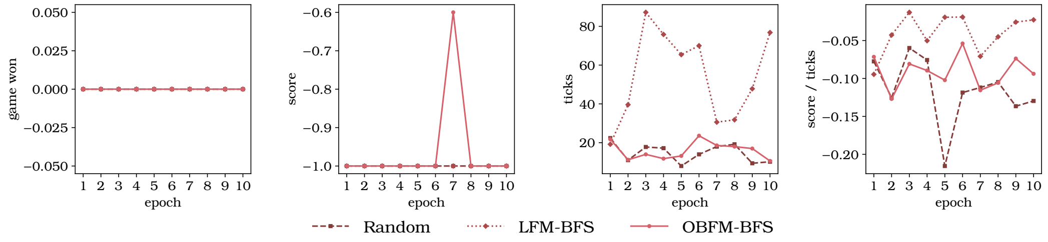


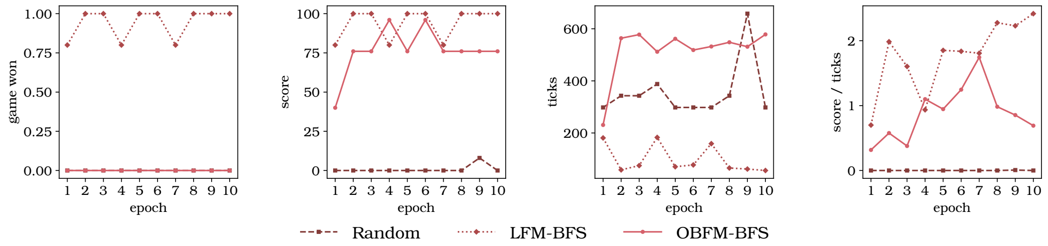
Figure A.3: Average performance per epoch in the continuous learning evaluation.



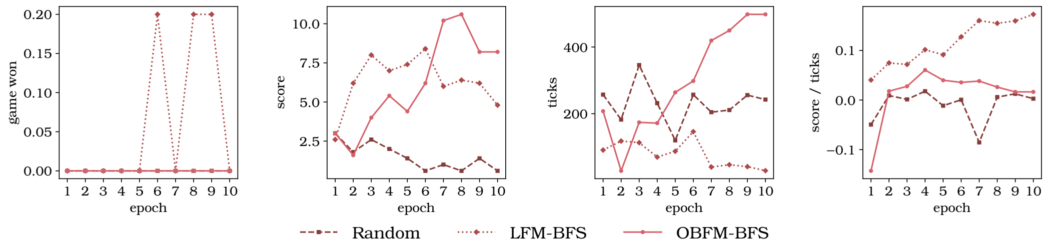
f) escape



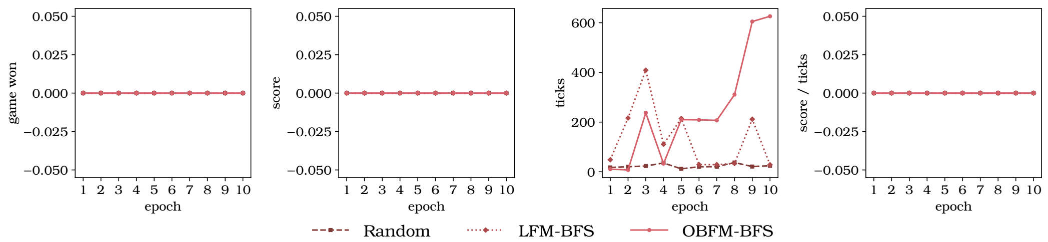
g) garbagecollector



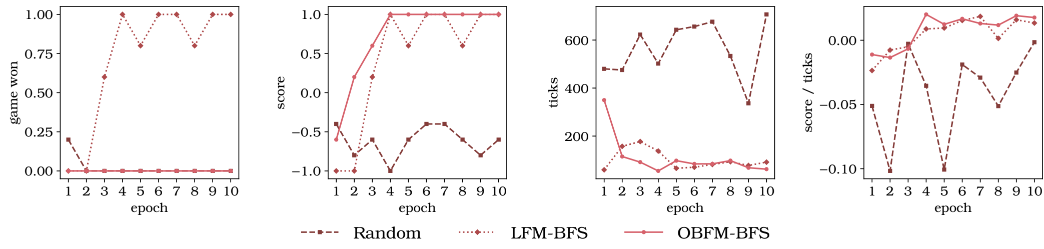
h) hungrybirds



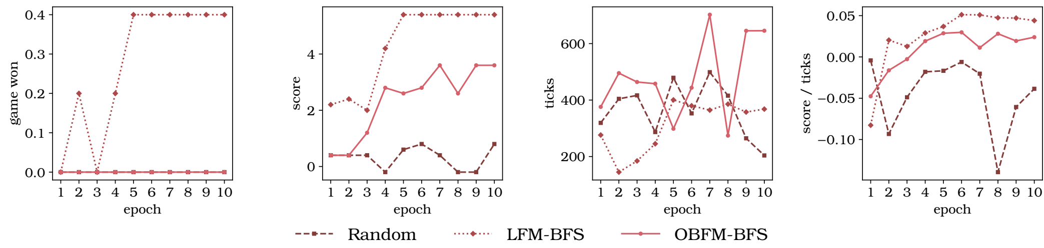
i) iceandfire



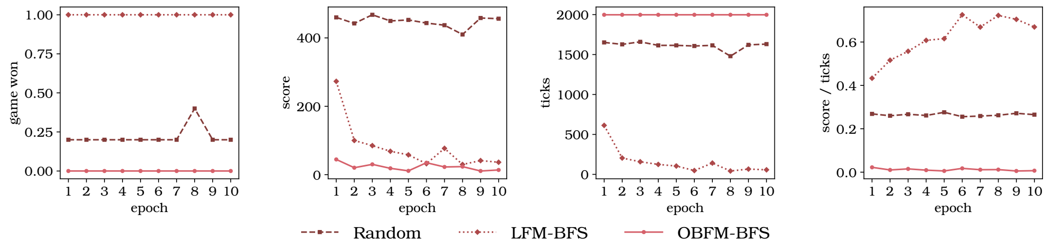
j) islands



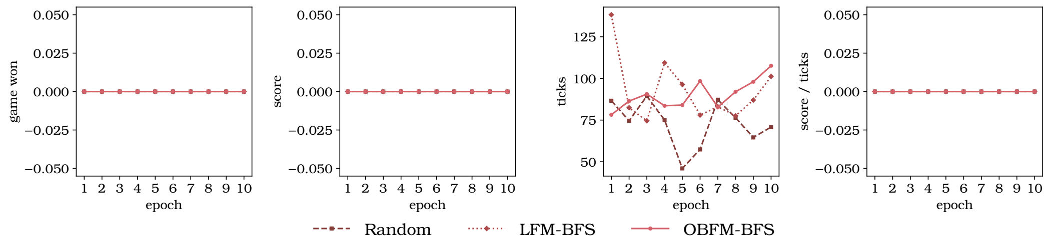
k) labyrinth



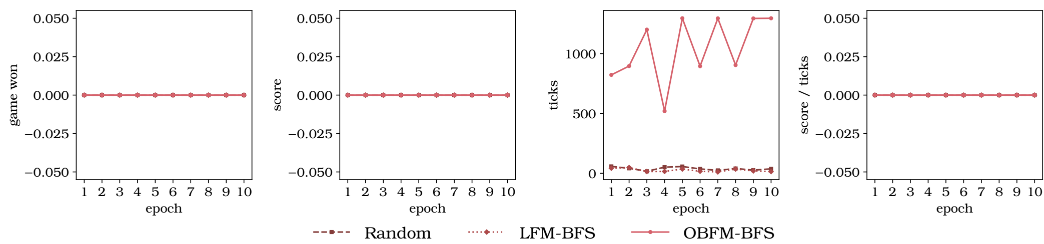
l) labyrinthdual



m) painter



n) run



o) watergame

A.6 Transfer Learning Game-Playing Performance

Table A.48: Transfer learning performance - bait

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	2.25	111.33	427.3	1st
LFM	BFS	0.0	1.0	—	1201.65	3rd
OBFM	BFS	0.0	1.5	—	1118.35	2nd

Table A.49: Transfer learning performance - catapults

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.25	—	10.9	3rd
LFM	BFS	0.0	0.5	—	219.75	1st
OBFM	BFS	0.0	0.45	—	1199.85	2nd

Table A.50: Transfer learning performance - chipschallenge

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.5	—	418.9	1st
LFM	BFS	0.0	0.0	—	669.85	3rd
OBFM	BFS	0.0	0.0	—	1250.65	2nd

Table A.51: Transfer learning performance - clusters

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-0.78	—	58.15	3rd
LFM	BFS	0.0	-0.2	—	982.6	2nd
OBFM	BFS	0.0	0.0	—	749.5	1st

Table A.52: Transfer learning performance - decepticoins

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	1.5	667.78	326.65	3rd
LFM	BFS	0.05	1.05	206.0	84.63	2nd
OBFM	BFS	0.5	0.5	58.1	9.0	1st

Table A.53: Transfer learning performance - escape

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-1.0	—	117.96	3rd
LFM	BFS	0.0	-0.4	—	737.55	2nd
OBFM	BFS	0.0	0.0	—	998.0	1st

Table A.54: Transfer learning performance - garbagecollector

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-1.0	—	14.72	2nd
LFM	BFS	0.0	-1.0	—	5.4	3rd
OBFM	BFS	0.0	-0.5	—	30.65	1st

Table A.55: Transfer learning performance - hungrybirds

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	2.0	—	356.5	3rd
LFM	BFS	1.0	100.0	100.2	—	1st
OBFM	BFS	0.25	25.0	115.6	298.0	2nd

Table A.56: Transfer learning performance - iceandfire

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.55	—	230.78	3rd
LFM	BFS	0.0	2.45	—	329.55	1st
OBFM	BFS	0.0	1.45	—	498.0	2nd

Table A.57: Transfer learning performance - islands

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	0.0	—	23.1	2nd
LFM	BFS	0.0	0.0	—	14.9	3rd
OBFM	BFS	0.0	0.0	—	503.45	1st

Table A.58: Transfer learning performance - labyrinth

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-0.85	462.0	565.31	3rd
LFM	BFS	0.0	-0.05	—	948.7	2nd
OBFM	BFS	0.05	-0.9	172.0	99.42	1st

Table A.59: Transfer learning performance - labyrinthdual

Agents		average win-rate	average score	average ticks won	average ticks lost	Rank
Random		0.0	-0.65	—	364.16	2nd
LFM	BFS	0.0	-0.1	—	601.85	1st
OBFM	BFS	0.0	-1.0	—	18.35	3rd

Table A.60: Transfer learning performance - painter

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.5	244.85	245.64	1998.0	2nd
LFM	BFS	0.25	256.05	227.0	1998.0	3rd
OBFM	BFS	0.5	285.7	469.3	1998.0	1st

Table A.61: Transfer learning performance - run

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	0.0	—	72.8	2nd
LFM	BFS	0.0	0.0	—	69.55	3rd
OBFM	BFS	0.0	0.0	—	93.65	1st

Table A.62: Transfer learning performance - watergame

Agents		average win-rate	average score	average ticks won	lost	Rank
Random		0.0	0.0	—	38.62	3rd
LFM	BFS	0.0	0.0	—	116.2	2nd
OBFM	BFS	0.0	0.0	—	609.2	1st

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den 13.12.2019

Alexander Dockhorn