AN ORDINAL AGENT FRAMEWORK

TOBIAS JOPPEN

Dissertation zur Erlangung des
akademischen Grades *Doktor-Ingenieur* (Dr.-Ing.)

Dissertationsschrift in englischer Sprache von
M.Sc Tobias Joppen
aus Darmstadt, Deutschland
geb. am 26.03.1989 in Bad Berleburg

Erstreferent: Prof. Dr. Kristian Kersting
Korreferent: Prof. Dr. Johannes Fürnkranz

Tag der Einreichung: 03.02.2021
Tag der Prüfung: 19.03.2021

Knowledge Engineering Group
Technische Universität Darmstadt
Darmstadt, 2021

02.02.2021 – Version 1.0

Tobias Joppen: *An Ordinal Agent Framework*.

Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung auf TUprints: 2021
URN: urn:nbn:de:tuda-tuprints-197490
Tag der mündlichen Prüfung: 19.03.2021

Abstract

In this thesis, we introduce algorithms to solve ordinal multi-armed bandit problems, Monte-Carlo tree search, and reinforcement learning problems. With ordinal problems, an agent does not receive numerical rewards, but ordinal rewards that cope without any distance measure. For humans, it is often hard to define or to determine exact numerical feedback signals but simpler to come up with an ordering over possibilities. For instance, when looking at medical treatment, the ordering *patient death*$\prec$*patient ill*$\prec$*patient cured* is easy to come up with but it is hard to assign numerical values to them.

As most state-of-the-art algorithms rely on numerical operations, they can not be applied in the presence of ordinal rewards. We present a preference-based approach leveraging dueling bandits to sequential decision problems and discuss its disadvantages in terms of sample efficiency and scalability. Following another idea, our final approach to identify optimal arms is based on the comparison of reward distributions using the Borda method. We test this approach on multi-armed bandits, leverage it to Monte-Carlo tree search, and also apply it to reinforcement learning. To do so, we introduce a framework that encapsulates the similarities of the different problem definitions. We test our ordinal algorithms on frameworks like the General Video Game Framework (GVGAI), OpenAI, or synthetic data and compare it to ordinal, numerical, or domain-specific algorithms. Since our algorithms are time-dependent on the number of perceived ordinal rewards, we introduce a binning method that artificially reduces the number of rewards.

Zusammenfassung

Diese Arbeit beschäftigt sich mit ordinalen Agenten im Rahmen von mehrarmigen Banditenproblemen, Monte-Carlo Baumsuche und verstärktem Lernen. Dabei liegt das Hauptaugenmerk auf der Einführung neuer Algorithmen, die in der Lage sind ordinale Belohnungen zu verarbeiten. Diese besitzen im Gegensatz zu numerischen Belohnungen kein Distanzmaß, was das Erstellen von Belohnungen für den Menschen oft vereinfacht. Ein beliebtes Beispiel kommt aus dem Bereich der ärztlichen Behandlung, wo die folgenden Konsequenzen leicht in eine Ordnung gebracht werden können: *Patient stirbt*$\prec$*Patient krank*$\prec$*Patient geheilt*. Den einzelnen Konsequenzen einen schlüssigen numerischen Wert zuzuordnen fällt eher schwer.

In dieser Arbeit verfolgen wir zunächst die Idee, Ergebnisse aus dem Bereich der duellierenden Banditen auf sequentielle Entscheidungsprobleme zu übertragen, was aber eine nicht effiziente Verwendung von Belohnungen nach sich zieht. Der stattdessen in dieser Arbeit verwendete Hauptansatz baut auf Wahlsystemen auf (genauer: die Borda Methode) um unterschiedliche Verteilungen von ordinalen Belohnungen zu vergleichen und die beste Option im Stil einer Wahl zu identifizieren. Dazu führen wir ein neues Framework ein, das die unterschiedlichen Problemstellungen vereinheitlicht. Dieser Ansatz wird zunächst auf mehrarmigen Banditenproblemen getestet, folgend auf Monte-Carlo Baumsuche erweitert und weiterhin auf verstärktes Lernen angewendet. Dabei verwenden wir sowohl existierende Testframeworks, wie das General Video Game Framework (GVGAI) und OpenAI als auch synthetische Experimente. In den Experimenten vergleichen wir unsere neuen Algorithmen mit schon existierenden ordinalen Ansätzen, numerischen Algorithmen oder domänenspezifischen Algorithmen wenn möglich. Da unsere neu eingeführten Algorithmen laufzeittechnisch von der Anzahl der unterschiedlichen ordinalen Belohnungen abhängen, stellen wir eine Binning Methode vor, die die Anzahl der Belohnungen künstlich reduziert.

Publications

Parts of this thesis have been previously published in the following publications:

[1] Tobias Joppen and Johannes Fürnkranz. "Ordinal Monte Carlo tree search." In: *Proceedings of the AIII IJCAI Workshop on Monte Carlo Search (MCS)*. 2020.

[2] Tobias Joppen, Tilman Strübig, and Johannes Fürnkranz. "Ordinal bucketing for game trees using dynamic quantile approximation." In: *Proceedings of the IEEE Conference on Games (CoG)*. IEEE. 2019, pp. 1–8.

[3] Tobias Joppen, Christian Wirth, and Johannes Fürnkranz. "Preference-based Monte Carlo tree search." In: *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI)*. Springer. 2018, pp. 327–340.

[4] Alexander Zap, Tobias Joppen, and Johannes Fürnkranz. "Deep ordinal reinforcement learning." In: *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*. Springer. 2019, pp. 3–18.

Other topic related publications that have not been included into this thesis:

[1] Tobias Joppen. "Monte Carlo tree search without numerical rewards." In: *From Multiple Criteria Decision Aid to Preference Learning (DA2PL)* (2018).

[2] Tobias Joppen, Miriam U Schröder, Nils Schröder, Christian Wirth, and Johannes Fürnkranz. "Informed hybrid game tree search for general video game playing." In: *IEEE Transactions on Games* 10.1 (Mar. 2018), pp. 78–90.

[3] Maryam Tavakol, Tobias Joppen, Ulf Brefeld, and Johannes Fürnkranz. "Personalized transaction kernels for recommendation using MCTS." In: *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI)*. Springer. 2019, pp. 338–352.

Contents

List of Figures

List of Tables

Acronyms

AI *Artificial Intelligence*

ANN *Artificial Neural Network*

DNN *Deep Neural Network*

DB *Duelling Bandits*

UCB *Upper Confidence Bound*

QUCB *Qualitative Upper Confidence Bound*

UCT *Upper Confidence Bound for Trees*

RL *Reinforcement Learning*

PL *Preference Learning*

OAF *Ordinal Agent Framework*

MCTS *Monte Carlo Tree Search*

B-MCTS *Borda Monte Carlo Tree Search*

H-MCTS *Heuristic Monte Carlo Tree Search*

PB-MCTS *Preference-Based Monte Carlo Tree Search*

MDP *Markov Decision Process*

MAB *Multi-Armed Bandit*

OMAB *Ordinal Multi-Armed Bandit*

MDPP *Markov Decision Process with Preferences*

PBB *Preference-Based Bandit*

DQL *Deep Q-Learning*

DQN *Deep Q-Network*

1 Introduction

The creation of intelligently behaving computer systems is a core problem tackled in the domain of *Artificial Intelligence* (AI). With *Intelligence* being a vaguely defined term, this domain covers many different problem settings with even more approaches to solve them.

Many modern AI problems can be described as a *Markov Decision Process* (MDP), where it is required to select the best action in a given state, to achieve a long term goal like reaching a certain state or maximizing a reward. *Monte Carlo Tree Search* (MCTS) is a popular technique for determining the best actions in MDPs [7, 23], and is a core component in many successful AI systems, such as AlphaGo [36], where it was the first algorithm to compete with professional players in this domain [24, 36]. MCTS is especially useful if no state features are available and strong time constraints exist, like in general game playing [11] or for opponent modeling in poker [33].

Classic MCTS depends on a numerical feedback or reward signal, as assumed by the MDP framework, where the algorithm tries to maximize the expectation of this reward. However, for humans, it is often hard to define or to determine exact numerical feedback signals. A suboptimally defined reward may allow the learner to maximize its rewards without reaching the desired extrinsic goal [2] or may require a predefined trade-off between multiple objectives [22]. This problem is particularly striking in settings where the natural feedback signal is inadequate to steer the learner to the desired goal. For example, if the problem is a complex navigation task and a positive reward is only given when the learner arrives in the goal state, the learner may fail because it will never find the way to the goal, and may thus never receive feedback from which it can improve its state estimations. A common concept to overcome this problem is *reward shaping*, the task of creating or changing a reward function [28]. As the rewards are used as direct feedback to achieve behavior that optimizes the aggregation of received rewards, the reward function has a significant impact on the behavior that is learned by the algorithm. Manual creation of the reward function is often expensive, non-intuitive, and difficult in certain domains and can therefore cause a bias in the optimal behavior that is learned by an algorithm. Since the learned behavior is sensitive to these reward values, the rewards of an environment should not be introduced or shaped arbitrarily if they are not explicitly known or naturally defined. This can also lead to another problem called *reward hacking*, where algorithms can exploit a reward function and miss the intended goal of the environment, caused by being able to receive better rewards through undesired behavior. Furthermore, the use of numerical rewards requires *infinite rewards* to model undesired deci-

sions to not allow trade-offs for a given state. This can be illustrated by an example in the medical domain, where it is undesirable to be able to compensate one occurrence of *death of a patient* with multiple occurrences of *a cured patient* to stay at a positive reward on average and therefore artificial feedback signals are used that can not be averaged. These issues have motivated the search for alternatives, such as preference-based feedback signals [46]. The idea behind preference-based signals and *Preference Learning* (PL) is to make decisions based on preference information. Often, it is easier for humans to generate consistent preference information than to come up with meaningful numerical values to compare different options [38]. For example, many people can easily tell whether they like something over an alternative or not, but it is getting hard when they are asked for a meaningful distance:

> How much more do you like pasta over pizza? Is that as much as you like green over blue?

Even though it might be easier for humans to generate preferences, they contain less information that an algorithm can use. A preference like $a \prec b$ does not tell about how a or b relate to c. This is a major disadvantage of PL, often referred to as sample efficiency.

Preference Learning

In this thesis, we investigate the use of rewards on an ordinal scale, where we have information about the relative order of various rewards, not about the magnitude of the quality differences between different rewards. Our goal is to extend agent algorithms with a focus on Monte Carlo Tree Search so that they can make use of ordinal rewards as an alternative feedback signal type to avoid and overcome the problems with numerical rewards. As ordinal feedback comes with less information and a less intuitive way to compare feedback in comparison to numerical rewards - especially the absence of a distance metric - we take a closer look at the imposed disadvantages of our novel ordinal algorithms in this thesis.

1.1 Research Challenges

In this thesis, we tackle the previously neglected but intuitive idea to create a preference-based or ordinal Monte Carlo Tree Search (MCTS) algorithm. MCTS provides real-time decision making for sequential decision problems and is heavily based and inspired on non-sequential decision problems. In non-sequential decision problems, you only need to make one decision that is not followed up by further decisions. Compare picking a lunch once with iteratively choosing the correct direction at intersections of a path: Choosing your lunch could be a choice between *sandwich, cereal, or croissant*. When going home from work, you repeatedly choose between different pathways at junctions. The different options one as at a time are called *arms* or *actions*. The area of non-sequential decision problems is well researched and ranges from common numeric algorithms like UCB1 to preference-based

algorithms that can function with ordinal feedback. The main research challenge throughout this thesis is to get an ordinal MCTS algorithm.

We formulate three Research Questions, as to be answered within this thesis:

1. Can we develop an MCTS solution that works with ordinal feedback?
2. Can we circumvent a bad sample efficiency?
3. Can we transform the solution of 1) to Reinforcement Learning?

1.2 Contributions and Outline

The content of this thesis is structured in multiple chapters as shown in Figure 1.1.

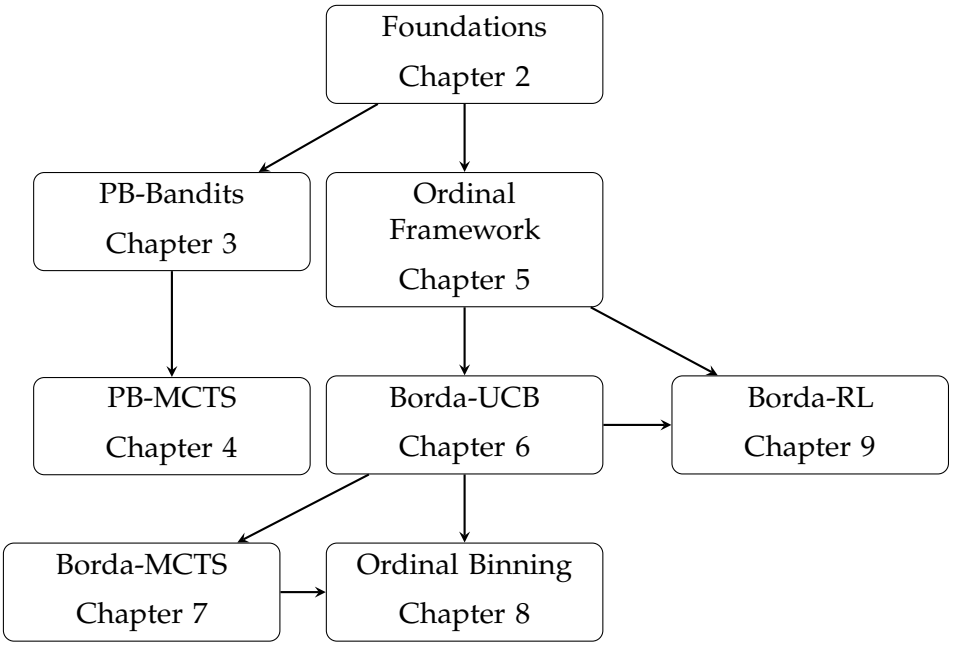


Figure 1.1: An overview of the upcoming chapters

Chapter 2 introduces background information and forms the foundation for the following chapters. Here, common concepts like *Artificial Neural Network* (ANN), MDP and *Reinforcement Learning* (RL) are introduced.

In Chapter 4, we tackle the first research question and present our first ordinal MCTS algorithm. It is based upon the concept of Dueling-Bandits that has been applied to tree search. The original paper[20] has been published at KI2018 where it won the *Best Paper Award*. The PB-MCTS algorithm does provide an answer to RQ1, but it has a bad sample efficiency. This algorithm functions as a proof of concept, but we argue that it is hard to increase the sample efficiency. Because

of this, the following sections are not based on this approach (see Figure 1.1).

In Chapter 5, we review the problems of PB-MCTS concerning the sample efficiency and introduce a new concept on how ordinal reviews can be utilized with a better sampling efficiency than PB-MCTS. The content of this chapter is novel such that it has not yet been published in this version. Rather, it summarizes, prepares, and introduces the core concept that the following chapters have in common. Here, we introduce the *Ordinal Agent Framework* (OAF) that exemplifies how ordinal agent algorithms can be designed. This chapter's main purpose is to draw a guiding thread throughout this thesis and show how Borda-MCTS, Borda-UCB, Borda-DQL, and Borda-QLearning are connected.

As the first algorithm that is based upon OAF, we introduce Borda-UCB in Chapter 6. Among all OAF implementations in this thesis, this is the most basic one solving the *Ordinal Multi-Armed Bandit* (OMAB) problem. The main idea of Borda-UCB is to store all seen rewards for different arms and to apply voting theory to identify good arms. Without knowledge of each other, we have developed Borda-UCB1 in parallel to another research group that managed to publish their results before us [17, 47]. We have abandoned our publication as we are able to base our ordinal MCTS algorithm on their publication, too. Our discontinued preprint containing Borda-UCB1 indicates the parallel development [17].

Borda-UCB provides an important step stone for the Borda-MCTS algorithm, the tree and MCTS version based on Borda-UCB. Borda-MCTS is introduced in Chapter 7 and functions as the core answer to research questions 1 and 2. The original paper is to be published at ICJAI2020 as part of the *Monte Carlo Search 2020* Workshop. A preprint is accessible [16].

As Borda-MCTS has a better sample-efficiency than PB-MCTS, it still scales with the number of seen disjunct ordinal rewards. One could easily come up with a problem where each reward is different from all previous ones, which would result in a very bad runtime for Borda-MCTS. To tackle this problem, we introduce an online technique to bucket ordinal rewards. As the main motivation, we want to limit the number of unique ordinal rewards that the Borda-Score needs to store since a high number of unique rewards could impair the performance of algorithms using the Borda-Score. Ordinal Bucketing is explained in Chapter 8 and has been published at the IEEE Conference on Games (CoG) 2019 [19].

Lastly, research question 3 is tackled in Chapter 9. By using the OAF, the concept used in Borda-UCB and Borda-MCTS can be applied to RL, leading to novel RL algorithms: We introduce Borda-QLearning and Borda-DQN as novel algorithms that can make use of ordinal rewards. By introducing those algorithms, we show that the idea of using Borda-Score as an exploitation term is not limited to the area of *Multi-Armed Bandit* (MAB) and MCTS, but can be applied to other AI

or optimization techniques, too. This chapter is based upon a paper published at ECML'19[53].

Finally, we conclude the thesis and revisit the research questions in Chapter 10.

2 Foundations

This chapter covers the foundations required to understand the contributions of this thesis. Besides basic principles, ideas, and definitions, seminal work from related domains is introduced. The purpose is to provide a basic insight into how these methods work, which will help to understand the more technical parts in the later sections. We cover *Artificial Neural Network* (ANN), *Multi-Armed Bandits* (MABs) of different kinds, *Markov Decision Process* (MDP), and *Reinforcement Learning* (RL). To each of these sections, we first will give a formal problem description followed by common methods to solve these problems. We focus on techniques that are relevant for understanding the main contributions of this thesis.

Thereafter, we review the concepts of *Markov Decision Processes* (MDPs), *Heuristic Monte Carlo Tree Search* (H-MCTS) and *Preference-Based Bandit* (PBB), which forms the basis of our work. We use an MDP as the formal framework for the problem definition, and H-MCTS is the baseline solution strategy we build upon. We also briefly recapitulate MAB as the basis of *Monte Carlo Tree Search* (MCTS) and their extension to *Preference-Based Bandit* (PBB).

2.1 Function Approximation

Function Approximation is the task of finding a function which approximates a given target function $U : D \mapsto C$ with any domain D and codomain C. There are different ways to define a function approximation problem. In this thesis, we are interested in the following problem setup:

Given information about the target function in pairs of $(d, U(d))$ with $d \in D$, the task is to choose a function V from a class of possible solutions $\mathscr{V}$. Thereby the main challenge is to pick a function such that the error on future predictions is low. To make *error* meaningful, one defines an error function on C:

$$E : C \times C \mapsto \mathbb{R}.$$

A popular choice for E on numerical codomains is the squared error $SE(d) = (U(d) - V(d))^2$ or the mean squared error when calculating the error over a set of data points $\hat{d} \subset D$: $MSE(\hat{d}) = \frac{1}{|\hat{d}|} \sum_{d \in \hat{d}} (U(d) - V(d))^2$.

One solution to the function approximation problem we use in this thesis are ANNs. In the following, we will introduce the basic concept of ANNs. Later, more advanced concepts are presented which are better suitable for the specific task we use them for.

2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a network of multi-layered artificial neurons. This network is composed of multiple layers: The first layer is called the input layer, the last is called the output layer with the rest being the hidden layers. A layer has a set of neurons, each having a set of weighted inputs, one output, and one activation function which maps the aggregated inputs to the output. The outputs of neurons in layer i are the input of neurons in layer $i+1$. The neuron inputs of the first layer are the ANNs inputs and the output of the last layers are the ANNs output. This way information can flow through the ANN from the inputs to the outputs. Usually, the structure of a neural network is a fixed hyperparameter that is not changed within the learning process. This is different from the weights parameter w which is changed by the learning process.

As a core part of brains, biological Neural Networks can learn and adapt their outputs to suit different inputs or situations. It is doing so by learning from example. The core idea of ANNs is to mimic this behavior: Given a set of known or historic input/output pairs, the task is to adjust the weights of the network to minimize the error of predicting the output of future input/output pairs. Taking a look at the previous Section 2.1, this is a function approximation. The main advantage of using an ANN for function approximation is that given a decent number of neurons and at least one hidden layer, an ANN can approximate every function. This does sound very powerful, but complexity always comes at a cost: Having many layers and neurons, the number of parameters (the input weights) grows. This is necessary for the ANN to be able to adapt to any function, but each weight needs to be fitted well until the complete ANN performs well. Thus an outstanding disadvantage of complex ANNs containing multiple layers (also called *Deep Neural Networks* (DNNs)) is the need of many training examples and training time [13].

2.2.1 Backpropagation

Being part of supervised learning, ANNs need to learn or adapt to new facts in an iterative fashion. At any given time t, ANNs have a concept of the world (the approximate function). At a later time, say at $t+1$, new facts about the world (samples of the target function) are known. These facts need to be integrated into the ANN to continuously learn and to converge towards the target function. Even though the right target value is known and one can compute the error of the network for new facts, there are no natural errors given for the hidden layer neurons. One popular technique is to propagate the output errors backwards through the network, leading to the so-called *Backpropagation* algorithm, which is composed of three steps[13]:

- The input values are set and a forward-pass calculates the approximate output values
- Given the correct output values, an error term is calculated
- The error is getting *backpropagated* through the network. Thereby the weights of each neuron are modified to reduce the error for future calculations of this input-output pair. Weights that had a bigger influence on the error receive a bigger change than those contributing less.

The *Forward Pass* defines how an ANN calculates an output estimate of input values and also the first step of the backpropagation algorithm. Here, the design of an ANN together with the current weight configuration w defines whether a neuron activates or not given its inputs. This is computed for each layer of neuron until the output layer.

Once a forward pass has calculated an estimate of the output neurons, the error of that estimation can be calculated using the true output values.

Given the error term for the output variables of a prediction, it is possible to measure the influence of each current weight on that error. Depending on how high that influence is, the current weights are adjusted to reduce the error of that prediction. This is done iteratively, starting at the output layer propagating that error back towards the input layer. The error of a hidden neuron thereby is an aggregation of errors of its succeeding neurons.

2.3 Markov Decision Process

In this thesis, we focus on a special subset of machine learning problems. All of these problems share a common internal problem structure: An *agent* (the machine learning algorithm) interacts with an *environment* (the world around it). Therefore, the agent has different *actions* that it can do (like walking forward, using objects, literally anything that it can do). At every time, the environment is in a certain *state*. This can be seen as a snapshot of the world at that time. When the agent acts, the environment changes, and the current state changes. Additionally, the agent has some kind of task. This can be reaching a special type of state (like winning a game or reaching a certain position) or maximizing a function (like walk as far to the east as possible or collect as many coins as possible). This is an iterative process (see Figure 2.1).

A *Markov Decision Process* (MDP) can be used to model these kinds of problems. It consists of the following:

- A (finite) set of *states* S.
- A (finite) set of *actions* A the agent can perform. Sometimes, only a subset $A(s) \subset A$ of actions is applicable in a state s.

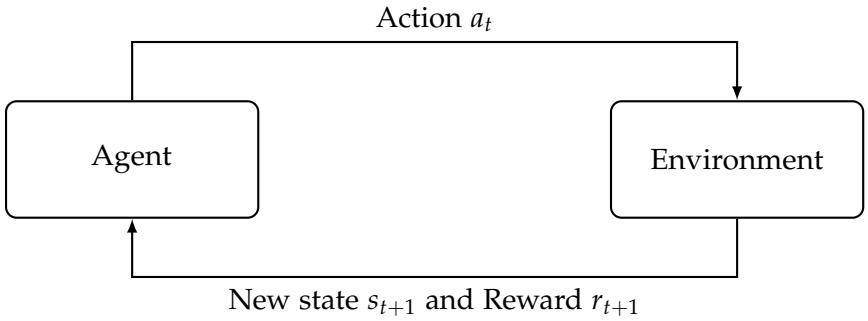


Figure 2.1: The Agent-Based Problem

- A Markovian *state transition* function $\delta(s' \mid s, a)$ denoting probability that invoking action a in state s leads the agent to state s'. If the state transition in state s playing action a is deterministic, we define $\delta(s, a) = s'$.
- A *reward function* $r(s)$ that defines a reward signal that the agent receives in state s. If the reward is not deterministic, the reward r is sampled from a reward distribution of the target state $P_s : R \mapsto \mathbb{R}$.
- A distribution of *start states* $\mu(s) \in [0, 1]$, defining the probability that the MDP starts in that state. We assume a single start state s_0, with $\mu(s_0) = 1$ and $\forall s \in S \setminus s_0 : \mu(s) = 0$.
- A set of *terminal states* for which $A(s) = \emptyset$. In some settings, we assume that only terminal states are associated with a non-zero reward.

A *policy* $\pi(a \mid s)$ defines the probability of selecting an action a in state s. As stated above, an agent has some kind of task and the target is to learn a policy π that fulfills this task as well as possible. What *as good as possible* exactly means depends on the concrete problem instance, but usually the task is to find the optimal policy $\pi^*(a \mid s)$ that maximizes some notion of value $V(\pi)$.

A very common way to define *as good as possible* is the cumulative reward

$$\begin{aligned} V(s_t) &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r s_t\right], \\ &= r(s_t) + \gamma \int_S \int_A \delta(s_{t+1} \mid a_t, s_t) \pi(a_t \mid s_t) V(s_{t+1}). \end{aligned} \tag{2.1}$$

Here, $\gamma \in [0, 1)$ is a discount factor, which dampens the influence of later events in the sequence. Here, the optimal policy $\pi^*(a \mid s)$ maximizes $V(s_t)$ for all time steps t.

Note the feedback type!

Note, that the cumulative reward function needs a numeric reward $r(s) \in \mathbb{R}$. Hence, ordinal feedback values can not be applied here directly. Later, in Section 5.2, we will introduce an ordinal value function. But since the majority of the following algorithms use numerical feedback, we stay with this definition for now.

In the following, we will introduce Multi-Armed Bandits, Monte-Carlo Tree Search, and the Reinforcement Learning Problem as instances of this problem definition. We start with Multi-Armed Bandits (MAB).

2.4 Multi-Armed Bandits

For finding an optimal policy, one needs to solve the so-called exploration/exploitation problem: The state/action spaces are usually too large to sample exhaustively. Hence, it is required to trade off the improvement of the current, best policy (exploitation) with an exploration of unknown parts of the state/action space.

The MAB problem is an MDP reduced to only one non-terminal state. Hence, exploration is only limited to the number of possible actions. This is a good limitation to start with, without a need of exploring a state space.

2.4.1 Common Definition

A *Multi-Armed Bandit* (MAB) is a finite sequential game between an agent and the environment. It is played in n rounds, which is called the time horizon. In each round t, the agent has to choose one action (or arm) $a_t \in A$ to play from a fixed set of actions A. Depending on the chosen arm a_t, the agent receives a reward for this turn r_t. In this work, we are interested in *stochastic stationary bandits* where the reward distribution per arm is constant over time and independent of whatever has happened in previous rounds. Hence, for now, MABs are a collection of reward probabilities: For each action $a \in A$, the probability distribution $P_a : R \mapsto \mathbb{R}$ defines how likely it is to see a reward r when playing a certain action a. For example, $P_{push}(1) \in [0, 1]$ defines how likely it is to gain a reward of 1 when using the action *push*.

Once the agent has chosen an arm a_t to play at time t, the environment samples the reward from the fixed distribution $r_t\tilde{}$

$a.The agent then can use this new reward r_t$ to choose which arm to pick next. In more detail, at time t, the agent knows the history of played arms $(a_0, a_1, ..., a_{t-2}, a_{t-1})$ and their rewards $(r_0, r_1, ..., r_{t-2}, r_{t-1})$. That historical information is used to decide which arm to play in the current round t. Further, for each action a, we introduce a multiset

$R_t(a)$ that keeps all rewards seen related to action a as well as t_a storing the number of times that action a has been played.

2.4.2 Multi-Armed Bandits as MDP

Markov Decision Process

The above section states that an MAB can be parameterized as $MAB(A, \{P_a\}_{a \in A})$ with A being the set of actions and P_a the reward distribution for action a. Since we have introduced MABs as a limited version of MDPs, we now show how an MAB instance $MAB(A, \{P_a\}_{a \in A})$ can be formalized as a MDP:

The MDP has the same set of actions $A = \{a_0, a_1, a_2, \dots\}$. It has a set of $|A| + 1$ states, composed of s_0 (the only non-terminal state with $A(s_0) = A$) and one state per action: $S = \{s_0, s_{a_0}, s_{a_1}, \dots\}$. At time t, the current state always is s_0, since all other states are terminal, leading to the only starting state s_0. The state transition function δ is deterministic, such that playing action a always leads to s_a. The reward an agent receives by reaching state s_a is sampled from P_a. This way we have created an MDP that resembles a given MAB.

2.4.3 Task Definition

Given a MAB, the task is to identify the best arm a^*. Usually, the *best arm* a^* is defined to be the one maximizing the average reward: $a^* = \arg\max_a \mathbb{E}[P_a]$.

Since the reward distributions of arms are unknown, the agent can not calculate the true average reward and simply pick the best arm. Instead, it has to figure it out manually by playing the arms and exploit the rewards. Intentionally neglecting any sense of optimality, a very simple approach would be to iteratively play all arms equally often and call the arm the best one who has the highest average of the perceived rewards. There are two main problems here:

First, imagine it is hard to differentiate the best from the second-best arm, but it is easy to separate those from the rest. Therefore, one would like to put more effort into testing those two good ones to get higher confidence of which one is the best, instead of pulling the arms equally often.

Second, once you have identified which arm is optimal, you do regret playing the others, since you could have done better in the past (and could have gotten better rewards). Hence, the common learning objective is to maximize the total reward $S_n = \Sigma_{t=1}^{n} r_t$.

In a mathematical sense, pulling an arm a involves a $regret(A) \mapsto \mathbb{R}_+$, where $regret(a^*) = 0$ since one does not regret playing the optimal arm. A commonly used regret is the expected reward difference compared to the best action:

$$regret(a) = \mathbb{E}(r_{a^*}) - \mathbb{E}(r_a). \tag{2.2}$$

Here it is $regret(a^*) = 0$ and $regret(a) > 0$ for non-optimal arms a. Notice that the term $\mathbb{E}(r_{a^*})$ from Formula 2.2 is independent of the

arm a and thus minimizing this regret is equivalent to maximizing the expected reward.

This notion of regret from ((2.2)) opens up the possibility to give insight into the worst-case performance of algorithms: Using probability theory, it is common to provide an upper bound to the accumulated regret of bandit algorithms.

The fundamental problem of solving MABs is the *Exploration-Exploitation Dilemma*: One wants to play the optimal arm a^* as often as possible. But to identify this arm, non-optimal arms do have to be played. In fact, one never knows for sure whether the arm with the currently highest sampled average reward is the optimal arm, or whether it was due to chance that the optimal one has a lower sampled value. Concluding, one wants to exploit the currently best arm to reach high rewards but also wants to explore the other arms to maybe find an even better arm to play.

2.4.4 The UCB1 Algorithm

After introducing the problem frameworks MAB and MDP, we introduce common algorithms to solve those. We start with a popular algorithm to solve MABs, where the task is to identify the arm (or action) with the highest return by repeatedly pulling one of the possible arms. In this setting, there is only one non-terminal state, and the task is to achieve the highest average reward by playing theoretically infinite times. Here the exploration/exploitation dilemma occurs because the player must play the best-known arm trying to maximize the average reward (exploitation), but also needs to search for the best arm among all alternatives (exploration). A well-known technique for resolving this dilemma in bandit problems is the *upper confidence bounds (UCB)* [3]. UCB estimates upper bounds on the expected reward for a certain arm and chooses the action with the highest associated upper bound. We then observe the outcome and update the bound. The simplest UCB policy (2.3) consists of two parts. The first term is $\bar{r}_{a,t}$, which is the average reward (given by all rewards of arm a until t: $\bar{r}_{a,t} = \sum_{r \in R_t(a)} \frac{r}{t}$). To this so called exploitation term (since it favors arms with high rewards), an exploration term $\sqrt{\frac{2 \ln t}{t_a}}$ is added. The term depends on the relative number of visits, increasing the chance that arms that have not yet been frequently played are selected in subsequent iterations.

$$UCB1 = \bar{r}_{a,t} + \sqrt{\frac{2 \ln t}{t_a}}. \tag{2.3}$$

The first term favors arms with high payoffs, while the second term guarantees exploration [3].

2.5 Monte-Carlo Tree Search

Markov Decision Process

Monte Carlo Tree Search (MCTS) is a method for approximating an optimal policy for a MDP. It builds a partial search tree, which is more detailed where the rewards are high. A node of the tree v corresponds to an environment state s_v. Having v_0 the root node, s_{v_0} is the environment state of which we want to get the optimal action choice. MCTS spends less time evaluating less promising action sequences but does not avoid them entirely to explore the state space. The algorithm iterates over four steps [7]:

1. *Selection:* Starting from the root node v_0, a *tree policy* traverses to deeper nodes v_k, until a node with unvisited successors is reached.
2. *Expansion:* One successor state is added to the tree.
3. *Simulation:* Starting from the new state, a so-called *rollout* is performed, i.e., random actions are played until a terminal state is reached or a depth limit is exceeded.
4. *Backpropagation:* The reward of the last state of the simulation is backed up through all selected nodes.

Given a random rollout policy, one only needs to define a tree policy that chooses the action worth playing in the tree. Starting at v_0 this is an iterative process over the nodes of the tree $v_0, v_1, \cdots$. At each node v_i, the tree policy needs to select one action. Since this is very similar to the bandit setting, a quite similar solution can be applied.

Multi-Armed Bandit

To do so, we handle each tree node as its own MAB problem and add a subscript v to formulae indicating the context. The UCT formula

$$a^*_{v,t} = \max_{a \in A(v)} \bar{r}_{v,t,a} + 2C\sqrt{\frac{2 \ln n_v}{n_v(a)}} \tag{2.4}$$

has been derived from the UCB1 algorithm (compare 2.3) by adding the contextual information v depicting the current node, and a newly introduced trade-off parameter C. In the following, we will often omit the subscript v or t when it is clear from the context.

Heuristic Monte Carlo Tree Search In large state/action spaces, rollouts can take many actions until a terminal state is observed. However, long rollouts are subject to high variance due to the stochastic sampling policy. Hence, it can be beneficial to disregard such long rollouts in favor of shorter rollouts with lower variance. *Heuristic MCTS* (H-MCTS) stops rollouts after a fixed number of actions and uses a heuristic evaluation function in case no terminal state was observed [7, 29, 31]. The heuristic is assumed to approximate $V(s)$ and can therefore be used to update the expectation.

2.6 Reinforcement Learning Problem

Reinforcement learning can be described as the task of learning a policy that maximizes the expected future numerical reward. In comparison to MCTS, RL learns a policy for each state s, where MCTS puts all effort into learning a policy for one action choice. To further depict the difference, RL does not need a forward model of the environment but learns from so-called episodes: Playthroughs of an MDP from a start state to a terminal state, like a chess game from start to end. An RL agent may perform multiple episodes until it finds a good policy, just like MCTS performs multiple iterations of its 4-step iteration. The RL agent learns iteratively by updating its current policy π after every action using the received reward from the environment. Using the previously defined formalism, this can be expressed as approximating the optimal policy iteratively with a function $\hat{\pi}$, by repeatedly choosing actions that lead to states s with the highest estimated value function $V_{\hat{\pi}}(s)$. In the following section, two common reinforcement learning algorithms are introduced.

2.6.1 Q-Learning

The key idea of the Q-learning algorithm [42] is to estimate Q-values $Q(s,a)$, which estimate the expected future sum of rewards $\mathbb{E}[R]$ when choosing an action a in a state s and following the optimal policy π^* afterward. Hence, the Q-value can be seen as a measure of *goodness* for a state-action pair (s,a), and therefore, in a given state s, the optimal policy π^* should select the action a that maximizes this value in comparison to other available actions in that state. The approximated Q-values are stored and iteratively updated in a Q-table. The Q-table is updated after an action a has been performed in a state s and the reward r and the newly reached state s' is observed. In a setting with deterministic state transitions, the computation of the expected Q-value is done by

$$\hat{Q}(s,a) = r(s') + \gamma \max_{a'} Q(s',a'), \tag{2.5}$$

where $s' = \delta(s,a)$ is the follow-up state reached when playing action a in state s. Following this so-called Bellman equation, every previously estimated Q-value is updated with the newly computed expected Q-value with the formula

$$Q(s,a) = Q(s,a) + \alpha[r(s') + \gamma \max_{a'} Q(s',a') - Q(s,a)], \tag{2.6}$$

where α represents a learning rate and γ the discount factor.

2.6.2 Deep Q-Network

The original Q-learning algorithm is limited to very simple problems, because of the explicitly stored Q-table, which essentially memorizes the quality of each possible state-action pair independently. Thus it requires, e.g., that each state-action pair has to be visited a certain number of times to make a reasonable prediction for this pair. A natural extension of this method is to replace the Q-table with a learned Q-function, which can predict a quality value for a given, possibly previously unseen state-action pair. At its very core, this is a Function Approximation task trying to learn a Q-function that resembles the environment rewards using an optimal policy (compare Section 2.1). As different approaches to approximate the Q-function are possible, the by far most popular and most researched option to learn Q-functions is the use of a DNN, leading to so-called *Deep Q-Network* (DQN). The key idea behind the DQN [26, 27] is to learn a continuous function $Q^{DQN}(s)$ in the form of a deep neural network with m_{in} input nodes, which represent the feature vector of s, and m_{out} output nodes, each containing the Q-value of one action a.

Deep Neural Network

Neural networks can be iteratively updated to fit the output nodes to the desired Q-values. The expected Q-value for a state-action pair is calculated in the same manner as defined in Formula 2.5 with the difference that the Q-values are now predicted by the DQN, with one output node $Q_a^{DQN}(s)$ for each possible action a. Therefore Formula 2.5 becomes

$$\hat{Q}_a^{DQN}(s) = r(s') + \gamma \max_{a'} \hat{Q}_{a'}^{DQN}(s') \tag{2.7}$$

where $\hat{Q}_a^{DQN}(s)$ represents the Q-value node of action a in state s.

In order to optimize the learning procedure, DQN makes use of several optimizations such as *experience replay*, the use of a separate *target and evaluation network*, and *Double Deep Q-Network*. More details on these techniques can be found in the following paragraphs.

Experience replay. Using a neural network to fit the Q-value of the previously executed state-action pair as described in Formula 2.7 leads to overfitting to recent experiences because of the high correlation between environmental states across multiple successive time steps, and the property of neural networks to overfitting recently seen training data. Instead of only using the previous state-action pair for fitting the DQN, experience replay [25] uses a memory M to store previous experience instances (s, a, r, s') and iteratively reuses a random sample of these experiences to update the network prediction at every time step.

Deep Q-Network

Target and evaluation networks. Frequently updating the neural network, which is simultaneously used for the prediction of the expected Q-value, leads to an unstable fitting of the network. Therefore these two tasks, firstly the prediction of the target Q-value for network fitting and secondly the prediction of the Q-value which is used for policy

computation, allows for a split into two networks. These two networks are the *evaluation network*, which is used for policy computation, and the *target network*, which is used for predicting the target value for continuously fitting the evaluation network. In order to keep the target network up to date, it is replaced by a copy of the evaluation network every c steps.

Double Deep Q-Network. Deep Q-Networks tend to overestimate the prediction of Q-values for some actions, which may result in an unjustified bias towards certain actions. To address this problem, Double Deep Q-Networks [40] additionally uses the target and evaluation networks to decouple the action choice and Q-value prediction by letting the evaluation network choose the next action to be played, and letting the target network predict the respective Q-value.

3 Preference-Based Bandits

In bandit problems, an agent solves a MAB by selecting the action that maximizes the reward. Most state-of-the-art algorithms assume numerical rewards. In domains like finance, a real-valued reward is naturally given, but many other domains do not have a natural numerical reward representation. In such cases, numerical values are often handcrafted by experts so that they optimize the performance of their algorithms. This process is not trivial, and it is hard to argue about good rewards. Hence, such handcrafted rewards may easily be erroneous and contain biases. For special cases such as domains with true ordinal rewards, it has been shown that it is impossible to create numerical rewards that are not biased. For example, [48] argues that emotions need to be treated as ordinal information.

Multi-Armed Bandit

In fact, it often is hard or impossible to tell whether domains are real-valued or ordinal by nature. Experts may even design handcrafted numerical rewards without thinking about alternatives, since using numerical rewards is state of the art and most algorithms need them. With this work, we want to emphasize that numerical rewards do not have to be the ground truth and it may be worth-while for the machine learning community to have a closer look at other options, too.

A popular example where the use of numerical values fails is a minimalistic medicine treatment setting: Consider three possible outcomes of treatment: healthy, unchanged, and dead. In the process of reward shaping, one assigns a numerical value to each outcome. Those numbers define the trade-off between how many patients have to be healed until one patient may die in comparison to no treatment (all patients unchanged). It is impossible to avoid this trade-off with numerical values. Using preferences or an ordinal score one could define the outcomes to be ordered as *healthy* $>$ *unchanged* $>$ *dead* without an implicit trade-off.

In this chapter, we introduce preference-learning, preference-based bandits, and a preference-based MDP variant called *Markov Decision Process with Preferences* (MDPP).

Markov Decision Process

3.1 Preference-Based Learning

In the past chapter, we have introduced MAB, MCTS, and RL as algorithms that solve MDPs - each with their own problem formulation. Those three problem formulations explicitly need the environment feedback to be numeric ($r(s) \in \mathbb{R}$). The research question of this work is directed towards preference-based and ordinal feedback. In this chapter, we focus on the concept of preference-based learning.

Monte Carlo Tree Search
Reinforcement Learning
Markov Decision Process

3.2 Learning Preferences

Looking at the algorithms introduced in Chapter 2, the best action a among a set of possible actions A is identified using its average reward or expected long-term reward. Here, the quality of each action is measured independent of its alternatives. The concept of preference learning shows an alternative to this approach: The core idea of preference learning is to utilize preference information about an object, label, action, or state pairs like (s_1, s_2), instead of scalar values. A preference can either be $s_1 \succ s_2$ indicating that s_1 performed better than s_2, $s_1 \prec s_2$ for s_2 beating s_1, $s_1 \sim s_2$ if both performed equally good or $s_1 \not\sim s_2$ if they can not be compared. Given a set of preference information, the target of a preference-based agent usually is to rank all available actions from best to worst or to identify the optimal action choice based on the given preferences.

3.3 MDP With Preferences

The MDPP is a MDP variant that utilizes preference feedback instead of numeric feedback. It is defined by $(S, A, \mu, zdeltas, \gamma, \rho)$ with S, A, μ, δ, and γ being the state space, action space, starting state distribution, state transition function, and discount factor as defined for the vanilla MDP in Section 2.3. The parameter ρ replaces the original reward signal $r(a, s)$ with a function denoting preference relations over trajectory pairs. A trajectory is defined as a sequence of states and actions: $\tau = \{s_0, a_0, s_1, a_1, \ldots, s_{n-1}, a_{n-1}, s_n\}$. As such, an agent does not receive a numerical reward for a single trajectory or action choice but receives a preference over two trajectories $\tau_i \succ \tau_j$. The function $\rho(\tau_i, \tau_j) \in [0, 1]$ is unknown to the agent and denotes the probability that the agent receives the preference $\tau_i \succ \tau_j$ when comparing the trajectory pair (τ_i, τ_j). The agent can sample ρ over time - usually receiving one preference (= reward) per time step t. This leads to the set of sampled preferences $\zeta = \{\zeta_i\} = \{\tau_{i1} \succ \tau_{i2}\}_{i=1,\ldots,t}$.

The objective for an agent is to identify the optimal policy π^* that maximally complies with the set of sampled preferences ζ. A preference $\tau_i \succ \tau_j$ is said to be satisfied if

$$\tau_i \succ \tau_j \Leftrightarrow P_\pi(\tau_i) > P_\pi(\tau_i),$$

with

$$P_\pi(\tau) = \mu(s_0)\Pi_{t=1}^{\tau}\pi(a_t|s_t)\delta(s_{t+1}|s_t, a_t)$$

being the probability of realizing trajectory τ with policy π.

3.4 Dueling Bandits

In Chapter 2, we have introduced MDPs as a sequence problem version of MABs. In this section, we introduce *Duelling Bandits* (DBs) as the

reduced bandit version of MDPPs using only one non-terminal state. The overall goal of a *Duelling Bandits Problem* is to find the best arm of a set of available arms A. In an iterative game with time horizon n, the agent chooses two arms from whose he receives noisy preference feedback of one arm over the other. Just as for MABs, we thereby assume that the probability of arm a_1 winning versus arm a_2 originates from an unknown independent random variable that is stationary over time. This probability of arm a_1 beating arm a_2 is written as $P(a_1 > a_2) = \epsilon(a_1, a_2) + \frac{1}{2}$, where $\epsilon(a_1, a_2) \in (-\frac{1}{2}, \frac{1}{2})$ is a measure of distinguishability between those two arms a_1 and a_2 [51]. We assume the existence of a total ordering over the set of arms A, for which $a > a' \rightarrow \epsilon(a, a') > 0$ is true. We denote the overall best arm as a^*[1].

Knowing the optimal arm, we now introduce two notions of regret for choosing the arm pair a_1 and a_2. The *strong regret* compares the worse of both arms to the optimal arm a^*:

$$strongRegret(a_1, a_2) = max\{\epsilon(a^*, a_1), \epsilon(a^*, a_1)\}. \tag{3.1}$$

The *weak regret* is less punishing and compares the better of the two arms with the optimal arm:

$$weakRegret(a_1, a_2) = min\{\epsilon(a^*, a_1), \epsilon(a^*, a_1)\}. \tag{3.2}$$

There are two further model assumptions on ϵ: The *strong stochastic transitivity* requires any triplet of arms $a_i > a_j > a_k$ to fulfil

$$\epsilon(a_i, a_k) \geq max\{\epsilon(a_i, a_j), \epsilon(a_j, a_k)\} \tag{3.3}$$

and the *stochastic triangle inequality* requires any triplet of arms $a_i > a_j > a_k$ to fulfil

$$\epsilon(a_i, a_k) \leq \epsilon(a_i, a_j) + \epsilon(a_j, a_k). \tag{3.4}$$

Those two requirements are important to clearly distinguish the optimal arm from non-optimal arms. The *strong stochastic transitivity* thereby allows generalizing a distinguishability between the optimal and second-best arm to a distinguishability between the optimal and every other arm. Stochastic triangle inequality captures the condition that the probability of a bandit winning (or losing) a comparison will exhibit diminishing returns as it becomes increasingly superior (or inferior) to the competing bandit.

There are two commonly used models that fulfill these requirements: The Bradley-Terry model uses a positive real value μ_i for each arm. The comparisons are made using

$$P(a_i > a_j) = \frac{\mu_i}{\mu_i + \mu_j} \tag{3.5}$$

The second model is a Gaussian model. Here, each arm a_i has its random variable X_i with variance 1 and mean μ_i. Doing a comparison, the arm sampling the higher value wins:

$$P(a_i > a_j) = P(X_i - X_j > 0). \tag{3.6}$$

Note, that $X_i - X_j$ $N(\mu_i - \mu_j, 2)$.

1 This element is given due to the existence of the maximum of the total order over the set of arms A

RUCB

The *relative UCB* algorithm (RUCB [54]) allows computing approximate, optimal policies for PB-MABs by computing the Condorcet winner, i.e., the action that wins all comparisons to all other arms. To this end, RUCB stores the number of times w_{ij} an arm i wins against another arm j and uses this information to calculate an upper confidence bound

$$u_{ij} = \frac{w_{ij}}{w_{ij} + w_{ji}} + \sqrt{\frac{\alpha \ln t}{w_{ij} + w_{ji}}}, \tag{3.7}$$

for each pair of arms. $\alpha > \frac{1}{2}$ is a parameter to trade-off exploration and exploitation and t is the number of observed preferences. These bounds are used to maintain a set of possible Condorcet winners[*Condorcet winner not introduced*]. If at least one possible Condorcet winner is detected, it is tested against its hardest competitor.

Several alternatives to RUCB have been investigated in the literature, but most PB-MAB algorithms are "first explore, then exploit" methods. They explore until a pre-defined number of iterations is reached, and start exploiting afterward. Such techniques are only applicable if it is possible to define the number of iterations in advance. But this is not possible to do for each node. Therefore we use RUCB in the following. For a general overview of PB-MAB algorithms, we refer the reader to [8].

3.5 Ordinal Multi-Armed-Bandits

(Move to Framework Chapter) Preference-based learning is an interesting alternative to real-valued feedback. One of the most outstanding points here is to obviate the need for real-valued reward shaping.

With dueling bandits, we have introduced a preference-based bandits problem. There, we have shown the Gaussian models, a popular option to derive the preference feedback from. Thereby, two random variables are sampled and compared. The arm with the higher sampled value wins the comparison. This approach is completely fine but very inefficient in the number of samples used: Imagine that in two iterations, the arms a_i vs. a_j and a_k vs a_l are being pulled. Now, the sampled values $r_i \sim X_i$ is compared with $r_j \sim X_j$ (k and l respectively). The sample inefficiency arises from the fact, that r_i is not compared with r_k or r_l. If the arms a_i and a_k are going to be played in a future iteration, new values for the comparison will be sampled.

The ordinal multi-armed bandit problem solves this issue: The main idea here is, that r_i could be compared to the samples of other arms, too. The arm a_i is not compared to a *single* arm, but it is compared with all arms at the same time. Hence, in each iteration of O-MAB one arm is pulled and an ordinal valued feedback value is returned. This value can be stored and compared to other values of other arms.

In a more formal mathematical way, an O-MAB is defined just like a common MAB as explained in Section 2.4. The only difference is the reward function. Instead of using a real-valued reward function, the O-MAB utilizes an ordinal feedback function $\mathit{reward} : S \mapsto O$, where O is a totally ordered set of ordinal rewards. Note, that any real-valued reward function $\mathit{reward} : S \mapsto \mathbb{R}$ can be turned into an ordinal one by omitting the distance function over $\mathbb{R}$ and only using its total order.

4 Preference-Based MCTS

Starting with this chapter, we introduce novel concepts and algorithms to answer the research questions of this thesis. In this chapter, we introduce the concept of *Preference-Based Monte Carlo Tree Search* (PB-MCTS) as a first algorithmically approach for an MCTS variant that makes use of ordinal rewards. This chapter is based upon our original publication [20].

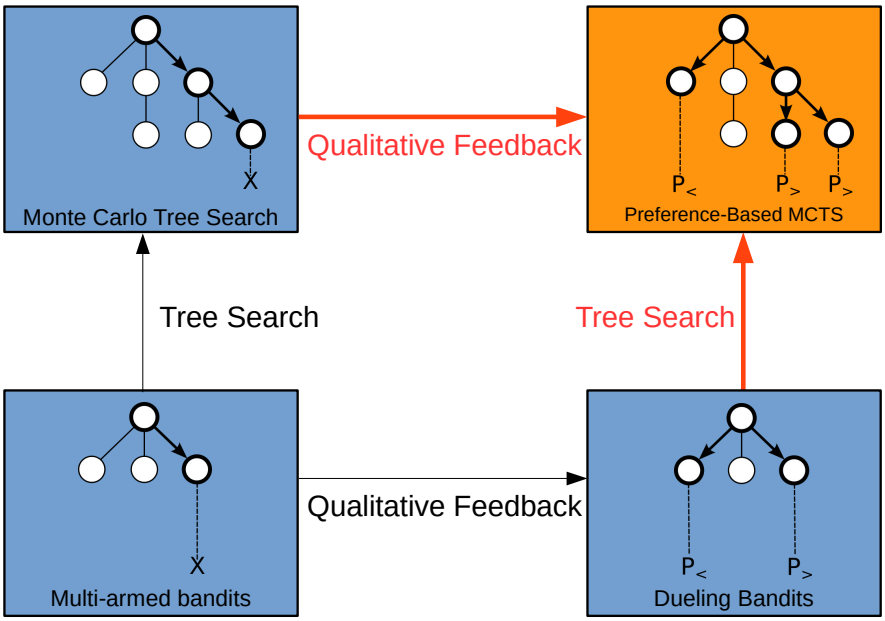


Figure 4.1: Research in Monte Carlo methods

4.1 Motivation

MCTS, especially H-MCTS, is often used in real-time scenarios like MDPs with time limitations. The rewards MCTS uses to estimate action values often are handcrafted. The definition of rewards might have been done by the environment creator, or, in case of H-MCTS, has been done by the creator of the heuristic used to estimate the quality of a state. In both cases, a human has defined the rewards.

Markov Decision Process

Are all rewards directly defined by humans?

There are other cases where rewards are not directly defined by humans, for example, if the reward measures a natural unit, the

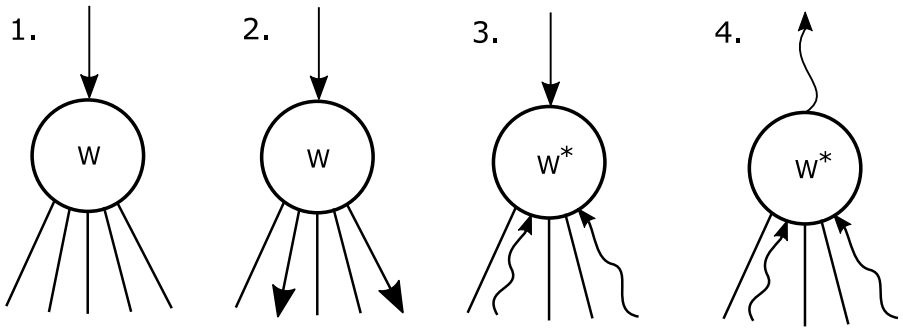


Figure 4.2: A local node view of PB-MCTS's iteration: selection; child selection; child backprop and update; backprop one trajectory.

occurrences of events, or if they are learned itself. For the latter, alpha zero, a very strong chess AI, is a popular example, where heuristic values are learned using a DNN.

Human-defined feedback does not always solely rely on the natural, *extrinsic* feedback from the domain. Furthermore, it often assumes an additional *intrinsic* feedback signal, which is comparable to the heuristic functions commonly used in classical problem-solving techniques. In this case, the learner may observe intrinsic reward signals for non-terminal states, in addition to the extrinsic reward in the terminal states. Ideally, this intrinsic feedback should be designed to naturally extend the extrinsic feedback, reflecting the expected extrinsic reward in a state, but this is often a hard task. If perfect intrinsic feedback is available in each state, making optimal decisions would be trivial. Hence heuristics are often error-prone and may lead to suboptimal solutions and agents may get stuck in local optima.

Deep Neural Network

4.2 Idea

The main research question of this thesis targets the creation of an ordinal MCTS algorithm. In this chapter, we present a first algorithm based on dueling bandits leaving the other research questions aside. Our new algorithm can be described as a preference-based variant of MCTS which works on *Markov Decision Process with Preferences* (MDPP) [**MDPP**], instead of vanilla MDPs. The basic idea behind the resulting preference-based Monte Carlo tree search algorithm is to use the principles of *preference-based* or *dueling bandits* [8, 50, 51] to replace the *multi-armed bandits* used in classic MCTS. Our work may thus be viewed as either extending the work on preference-based bandits to tree search or extending MCTS to allow for preference-based feedback, as illustrated in Fig. 4.1. Thereby, the tree policy does not select a single path, but a binary tree leading to multiple rollouts per iteration and we obtain pairwise feedback for these rollouts.

We evaluate the performance of this algorithm by comparing it to *Heuristic Monte Carlo Tree Search* (H-MCTS). Hence, we can deter-

Algorithm 1 One Iteration of PB-MCTS

function PB-MCTS $(\hat{S}, s, \alpha, \mathbf{W}, B)$
Input : A set of explored states $\hat{S}$, the current state s, exploration-factor α, matrix of wins $\mathbf{W}$ (per state), list of last Condorcet pick B (per state)
Output : $[s', \hat{S}, \mathbf{W}, B]$
$[a_1, a_2, B] \leftarrow$ SELECTACTIONPAIR$(\mathbf{W}_s, B_s)$
for $a \in \{a_1, a_2\}$ **do**
  $s' \sim \delta(s' \mid s, a)$
  if $s' \in \hat{S}$ **then**
    $[sim[a], \hat{S}, \mathbf{W}, B] \leftarrow$ PB-MCTS$(\hat{S}, s', \alpha, \mathbf{W}, B)$;
  else
    $\hat{S} \leftarrow \hat{S} \cup \{s'\}$
    $sim[a] \leftarrow$ SIMULATE(a)
  end
end
$w_{sa_1a_2} \leftarrow w_{sa_1a_2} + \mathbb{1}(sim[a_2] \succ sim[a_1]) + \frac{1}{2}\mathbb{1}(sim[a_1] \simeq sim[a_2])$
$w_{sa_2a_1} \leftarrow w_{sa_2a_1} + \mathbb{1}(sim[a_1] \succ sim[a_2]) + \frac{1}{2}\mathbb{1}(sim[a_2] \simeq sim[a_1])$
$s_{return} \leftarrow$ RETURNPOLICY$(s, a_1, a_2, sim[a_1], sim[a_2])$
return $[s_{return}, T, \mathbf{W}, B]$

mine the effects of approximate, heuristic feedback in relation to the ground truth. We use the 8-puzzle domain since simple but imperfect heuristics already exist for this problem.

This section can be viewed as an extension of previous work in two ways: (1) it adapts Monte Carlo tree search to preference-based feedback, comparable to the relation between preference-based bandits and multi-armed bandits, and (2) it generalizes preference-based bandits to sequential decision problems like MCTS generalizes multi-armed bandits.

4.3 Preference-Based MCTS

Just like *Upper Confidence Bound* (UCB) can be adapted to *Upper Confidence Bound for Trees* (UCT) for the application in trees (compare Section 2.5), this section describes our approach to translate RUCB into a tree version, as shown in Algorithm 1. In contrast to H-MCTS, PB-MCTS works for OMDPs and selects two actions per node in the *selection phase*, as shown in Fig. 4.2 and Fig. 4.3. Since RUCB is used as a tree policy, each node in the tree maintains its own weight matrix $\mathbf{W}$ to store the history of action comparisons in this node (as known

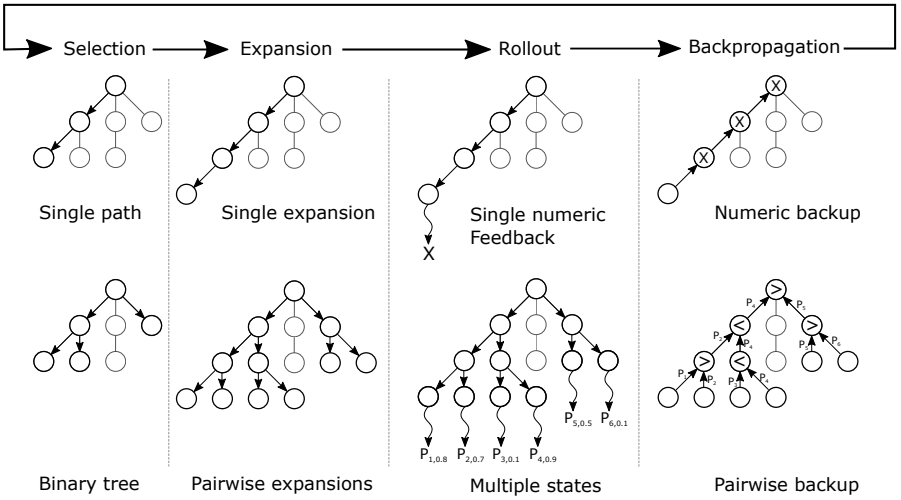


Figure 4.3: A comparison of iteration steps with backpropagation of MCTS (top, a path gets selected) and Preference-Based MCTS (bottom, a binary subtree gets selected).

from RUCB, compare Section 3.4). Actions are then selected based on a modified version of the RUCB formula (3.7)

$$\begin{aligned} \hat{u}_{ij} &= \frac{w_{ij}}{w_{ij}+w_{ji}} + c\sqrt{\frac{\alpha \ln t}{w_{ij}+w_{ji}}}, \\ &= \frac{w_{ij}}{w_{ij}+w_{ji}} + \sqrt{\frac{\hat{\alpha} \ln t}{w_{ij}+w_{ji}}}, \end{aligned} \tag{4.1}$$

where $\alpha > \frac{1}{2}$, $c > 0$ and $\hat{\alpha} = c^2\alpha > 0$ are the hyperparameters that allow to trade-off exploration and exploitation. Therefore, RUCB can be used in trees with the corrected lower bound $0 < \alpha$.

Based on this weight matrix, SELECTACTIONPAIR then selects two actions using the same strategy as in RUCB: If there is a Condorcet winner candidate ($C \neq \emptyset$), the first action a_1 is chosen among the possible Condorcet winners $C = \{a_c \mid \forall j : u_{cj} \geq 0.5\}$. Typically, the choice among all candidates $c \in C$ is random. However, in case the last selected Condorcet candidate in this node is still in C, it has a 50% chance to be selected again, whereas each of the other candidates can share the remaining 50% of the probability mass evenly. The second action a_2 is chosen to be a_1's hardest competitor, i.e., the move whose win rate against a_1 has the highest upper bound $a_2 = \arg\max_l u_{la_1}$. Note that, just as in RUCB, the two selected arms need not necessarily be different, i.e., it may happen that $a_1 = a_2$. This is a useful property because once the algorithm has reliably identified the best move in a node, forcing it to play a suboptimal move in order to obtain a new preference would be counter-productive. In this case, only one rollout is created and the node will not receive a preference signal in this node. However, the number of visits to this node is updated, which may lead to a different choice in the next iteration.

The *expansion and simulation phases* are essentially the same as in conventional MCTS except that multiple nodes are expanded in each iteration. SIMULATE executes the *simulation policy* until a terminal state or break condition occurs as explained below. In our experiments, the simulation policy performs a random choice among all possible actions. Since two actions per node are selected, one simulation for each action is conducted in each node. Hence, the algorithm traverses a binary subtree of the already explored state space tree before selecting multiple nodes to expand. As a result, the number of rollouts is not constant in each iteration but increases exponentially with the tree depth. The preference-based feedback is obtained from a pairwise comparison of the performed rollouts.

In the *backpropagation phase*, the obtained comparisons are propagated up towards the root of the tree. In each node, the **W** matrix is updated by comparing the simulated states of the corresponding actions a_i and a_j and updating the entry w_{ij}. Passing both rollouts to the parent in each node would result in an exponential increase of pairwise comparisons, due to the binary tree traversal. Hence, the newest iteration could dominate all previous iterations in terms of the gained information. This is a problem, since the feedback obtained in a single iteration may be noisy and thus yield unreliable estimates. Monte Carlo techniques need to average multiple samples to obtain a sufficient estimate of the expectation. Multiple updates of two actions in a node may cause further problems: The preferences may arise from bad estimates since one action may not be as well explored as the other. It would be unusual for RUCB to select the same two actions multiple times consecutively since either the first action is no Condorcet candidate anymore or the second candidate, the best competitor, will change. These problems may lead to unbalanced exploration and exploitation terms resulting in overly bad ratings for some actions. Thus, only one of the two states is propagated back to the root node. This way it can be assured that the number of pairwise comparisons in the nodes (and especially in the root node) remains constant ($= 1$) over all iterations, ensuring numerical stability.

For this reason, we need a *return policy* to determine what information is propagated upwards (compare RETURNPOLICY in Alg. 1). An obvious choice is the *best preference policy* (BPP), which always propagates the preferred alternative upwards, as illustrated in step four of Fig. 4.2. A random selection is used in case of indifferent actions. We also considered returning the best action according to the node's updated matrix **W**, to make a random selection based on the weights of **W**, and to make a completely random selection. However, preliminary experiments showed a substantial advantage when using BPP.

4.4 Experimental Setup

We compare PB-MCTS to H-MCTS in the 8-puzzle domain. The 8-puzzle is a move-based deterministic puzzle where the player can move numbers on a grid. It is played on a 3×3 grid where each of the 9 squares is either blank or has a tile with the number 1 to 8 on it.

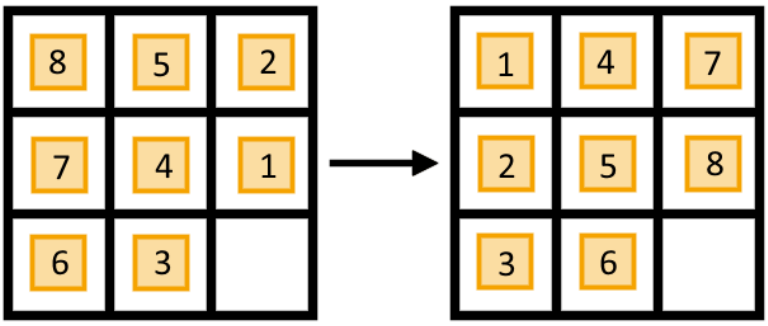


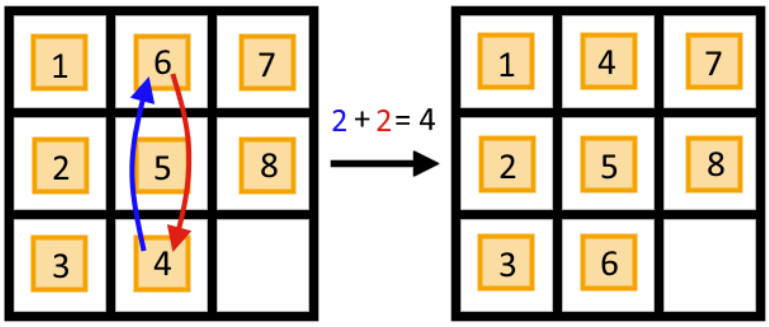
Figure 4.4: The start state (left) and end state (right) of the 8-Puzzle. The player can swap the positions of the empty field and one adjacent number.

A move consists of shifting one of the up to 4 neighboring tiles to the blank square, thereby exchanging the position of the blank and this neighbor. The task is then to find a sequence of moves that lead from a given start state to a known end state (see Fig. 4.4). The winning state is the only goal state. Since it is not guaranteed to find the goal state, the problem is an infinite horizon problem. However, we terminate the evaluation after 100 time-steps to limit the runtime. Games that are terminated in this way are counted as losses for the agent. The agent is not aware of this maximum.

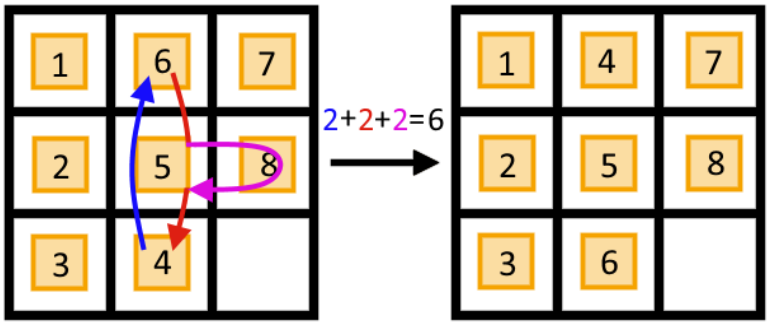
As a heuristic for the 8-puzzle, we use the *Manhattan distance with linear conflicts* (MDC), a variant of the *Manhattan distance* (MD). MD is an optimistic estimate for the minimum number of moves required to reach the goal state. It is defined as

$$H_{manhattan}(s) = \sum_{i=0}^{8} |pos(s,i) - goal(i)|, \tag{4.2}$$

where $pos(s,i)$ is the (x,y) coordinate of number i in game state s, $goal(i)$ is its position in the goal state, and $|\cdot|_1$ refers to the 1-norm or Manhattan-norm.



(a) Manhattan distance



(b) Manhattan distance with linear conflict

Figure 4.5: The two heuristics used for the 8-puzzle.

4.4.1 Heuristics

MDC additionally detects and penalizes linear conflicts. Essentially, a linear conflict occurs if two numbers i and j are on the row where they belong, but in swapped positions. For example, in Fig. 4.5b, the tiles 4 and 6 are in the right column but need to pass each other in order to arrive at their right squares. For each such linear conflict, MDC increases the MD estimate by two because in order to resolve such a linear conflict, at least one of the two numbers needs to leave its target row (1st move) to make place for the second number and later needs to be moved back to this row (2nd move). The resulting heuristic is still admissible in the sense that it can never over-estimate the actually needed number of moves.

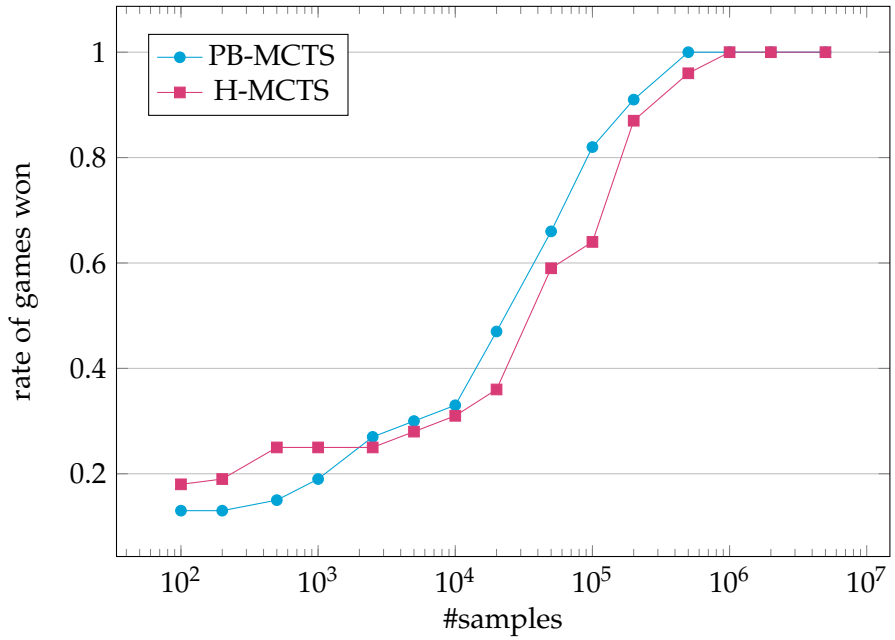


Figure 4.6: Using their best hyperparameter configurations, PB-MCTS and H-MCTS reach similar win rates.

4.4.2 Preferences

In order to deal with the infinite horizons during the search, both algorithms rely on the same heuristic evaluation function, which is called after the rollouts have reached a given depth limit. For the purpose of comparability, both algorithms use the same heuristic for evaluating non-terminal states, but PB-MCTS does not observe the exact values but only preferences that are derived from the returned values. Comparing arm a_i with a_j leads to terminal or heuristic rewards r_i and r_j, based on the according rollouts. From those reward values, we derive preferences

$$(a_k \succ a_l) \Leftrightarrow (r_k > r_l) \text{ and } (a_k \simeq a_l) \Leftrightarrow (r_k = r_l)$$

which are used as feedback for PB-MCTS. H-MCTS can directly observe the reward values r_i.

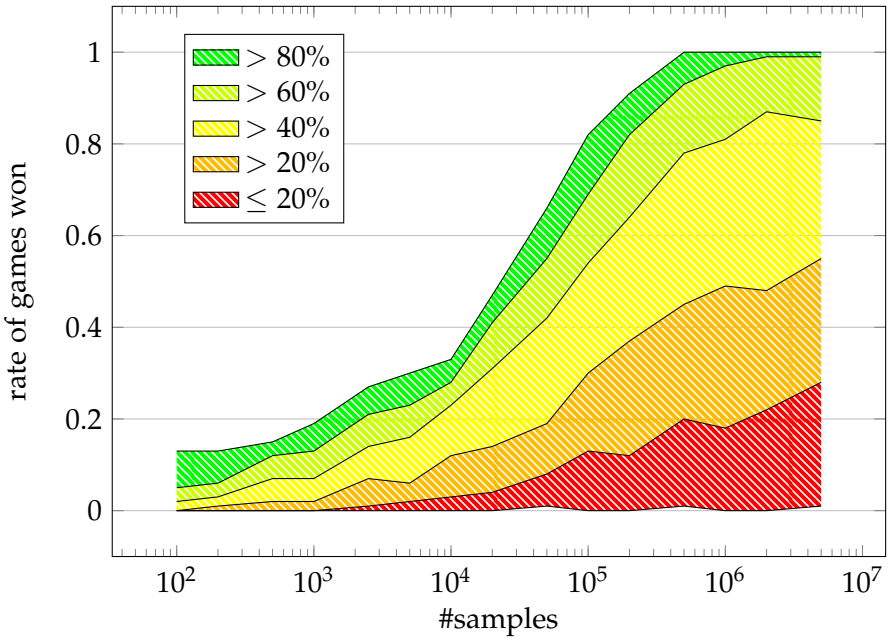


Figure 4.7: The configuration percentiles of PB-MCTS. Throughout all configurations, a robust curve without a steep decrease in win rate is shown.

4.4.3 Parameter Settings

Both algorithms H-MCTS and PB-MCTS are subject to the following hyperparameters:

- *Rollout length*: the number of actions performed at most per rollout (tested with: 5, 10, 25, 50).
- *Exploration-exploitation trade-off*: the C parameter for H-MCTS and the α parameter for PB-MCTS (tested with: 0.1 to 1 in 10 steps).
- *Allowed transition-function samples per move (#samples)*: a hardware-independent parameter to limit the time an agent has per move[1] (tested with a logarithmic scale from 10^2 to $5 \cdot 10^6$ in 10 steps).

For each combination of parameters, 100 runs are executed. We consider #samples to be a parameter of the problem domain, as it relates to the available computational resources. The rollout length and the trade-off parameter are optimized.

1 Please note that this is a fair comparison between PB-MCTS and H-MCTS: The first uses more #samples per iteration, the latter uses more iterations.

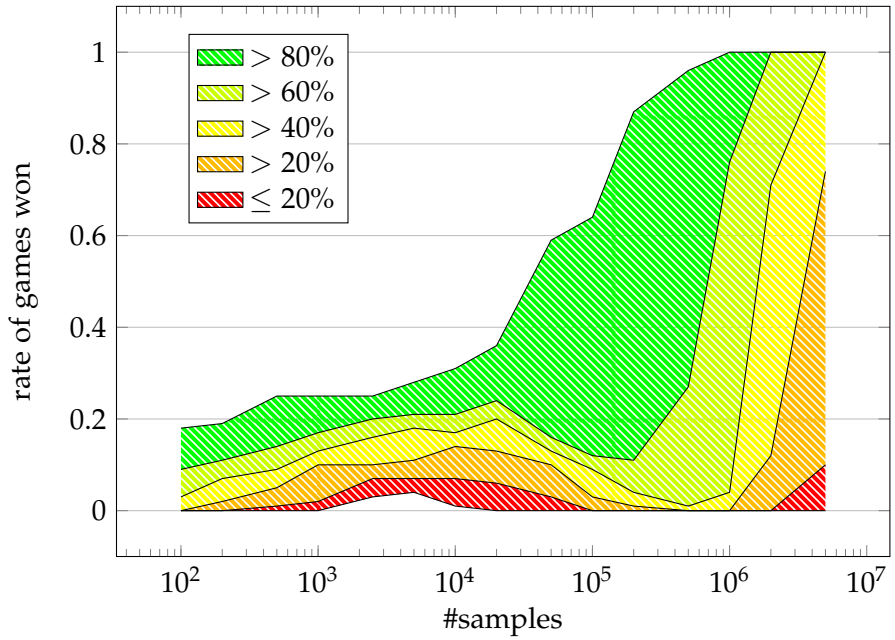


Figure 4.8: The configuration percentiles of H-MCTS. The distribution of hyperparameters to wins is shown in steps of 0.2 percentiles. The amount of wins decreases rapidly if the parameter setting is not among the best 20%.

4.5 Results

In the following, we present and discuss the results of the experiments shown above. First, we analyze the performance of both algorithms looking at the optimal configurations found. Then, the other configurations are considered, too, looking at how probable it is to find a good performing configuration.

4.5.1 Tuned: Maximal Performance

Fig. 4.6 shows the maximal win rate over all possible hyperparameter combinations, given a fixed number of transition-function samples per move. One can see that for a lower number of samples (≤ 1000), both algorithms lose most games, but H-MCTS has a somewhat better performance in that region. However, above that threshold, H-MCTS no longer outperforms PB-MCTS. On contrary, PB-MCTS typically achieves a slightly better win rate than H-MCTS.

4.5.2 Untuned: More robust but slower

We also analyzed the distribution of wins for non-optimal hyperparameter configurations. Fig. 4.7 and Fig. 4.8 shows several curves

of win rate over the number of samples, each representing a different percentile of the distribution of the number of wins over the hyperparameter configurations. The top lines of Fig. 4.7 and Fig. 4.8 correspond to the curves of Fig. 4.6, since they show the results of the optimal hyperparameter configuration. Below, we can see how non-optimal parameter settings perform. For example, the second line from the top shows the 80% percentile, i.e. the configuration for which 20% of the parameter settings performed better and 80% perform worse, calculated independently for each sample size. For PB-MCTS (Fig. 4.7), the 80% percentile line lies next to the optimal configuration from Fig. 4.6, whereas for H-MCTS there is a considerable gap between the corresponding two curves. In particular, the drop in the number of wins around $2 \cdot 10^5$ samples is notable. Apparently, H-MCTS gets stuck in local optima for most hyperparameter settings. PB-MCTS seems to be less susceptible to this problem because its win count does not decrease that rapidly.

On the other hand, untuned PB-MCTS seems to have a slower convergence rate than untuned H-MCTS, as can be seen for high #sample values. This may be due to the exponential growth of trajectories per iteration in PB-MCTS.

4.6 Conclusion

In this chapter, we proposed PB-MCTS, a new variant of Monte-Carlo tree search which is able to cope with preference-based feedback. In contrast to conventional MCTS, this algorithm uses relative UCB as its core component and is based upon a MDPP. We showed how to use trajectory preferences in a tree search setting by performing multiple rollouts and comparisons per iteration.

Upper Confidence Bound

Our evaluations in the 8-puzzle domain showed that the performance of H-MCTS and PB-MCTS strongly depends on adequate hyperparameter tuning. PB-MCTS is better able to cope with suboptimal parameter configurations and erroneous heuristics for lower sample sizes, whereas H-MCTS has a better convergence rate for higher values. In the tested environment, PB-MCTS seems to work well if tuned, but showing a more steady but slower convergence rate if untuned, which may be due to the exponential growth.

A major problem with preference-based tree search is the exponential growth in the number of explored trajectories: As long as the best arm has not been identified in a given node, two different actions are chosen for comparison. Having a tree depth of n leads to 2^n explored trajectories per iteration. Comparing to MCTS, PB-MCTS puts a lot of effort into exploring suboptimal choices, leaving the interesting actions less explored. Additionally, instead of aggregating rewards per action, PB-MCTS needs to aggregate rewards per action pairs. Hence, it also takes PB-MCTS more iterations to identify the best action than for

common MCTS. This reinforces the problem of selecting two actions, as it takes even more iterations to identify the best action.

This is a structural problem that is derived by using dueling bandits in a tree-based manner. One could try to research techniques that allow to prune the binary subtree - for example, motivated by alpha-beta pruning or similar techniques. But as this thesis is more focused on ordinal feedback, we follow a different approach by leaving the area of dueling bandits in the next chapter.

5 Ordinal Agent Framework

In Chapter 2 we have introduced three algorithms: UCB1, MCTS, and DQN. They all share a common concept of evaluating the quality of actions. Briefly speaking, actions are executed to receive feedback signals that indicate the quality of the actions. In more detail, looking at time step t, when playing action a_t in state s_t, the environment returns the new state s_{t+1} and a reward r_t that describes the quality of the state transition from s_t to s_{t+1} using a_t. As those are rewards sampled from an unknown distribution, multiple rewards are needed to be aggregated to yield a good estimation of the quality of that action. The estimated quality $v_t(a)$ is then used to determine which action to sample next. The exact action selection strategy differs between algorithms, but they all require the estimated quality v to be a real number. In *Multi-Armed Bandit* (MAB), the quality of an action is directly represented by its average reward $r_t(a)$ from the MDP. For MCTS it's slightly different, as not only the direct reward is used, but uses the reward generated by a rollout. MCTS and MAB both use the action-choosing policy, which is to choose $a_t = \arg\max_a v_t(a) + u_t(a)$ with $u_t(a)$ being an exploration term indicating the (un)certainty of $v_t(a)$ (See Section 2.4.4 and Section 2.5). In RL, we look at DQN here, things are a bit different: Instead of evaluating a complete rollout like in MCTS, quality values of follow-up states are reused using the Bellman equation and temporal difference learning. Here, a new reward sample of an action a is composed of the direct reward of the execution of that action $r_t(a)$ and the quality of the follow-up state that is reached by executing a.

Upper Confidence Bound
Monte Carlo Tree Search
Deep Q-Network

Reinforcement Learning
Deep Q-Network

Despite those differences in the reward generation (See Figure 5.1), all three algorithms follow the same abstract concept: in state s_t an action a_t is chosen based on the quality value $v_t(a)$ for each action available and some concept of exploration. The algorithm now executes action a_t, returning a reward r_t and the new state s_{t+1}. The last step, which we call the *aggregation*, uses the gathered information to update the quality values, especially $v_{t+1}(a_t)$.

The main approach for our new *Ordinal Agent Framework* (OAF) is to decouple the aggregation step from the rest of the algorithms. Our main target is to develop new aggregation steps that support ordinal feedback values. In the following chapters, we introduce new ordinal aggregation steps for MAB, MCTS, and DQNs. But first, we take a closer look at a more formal view of aggregation steps and our motivation.

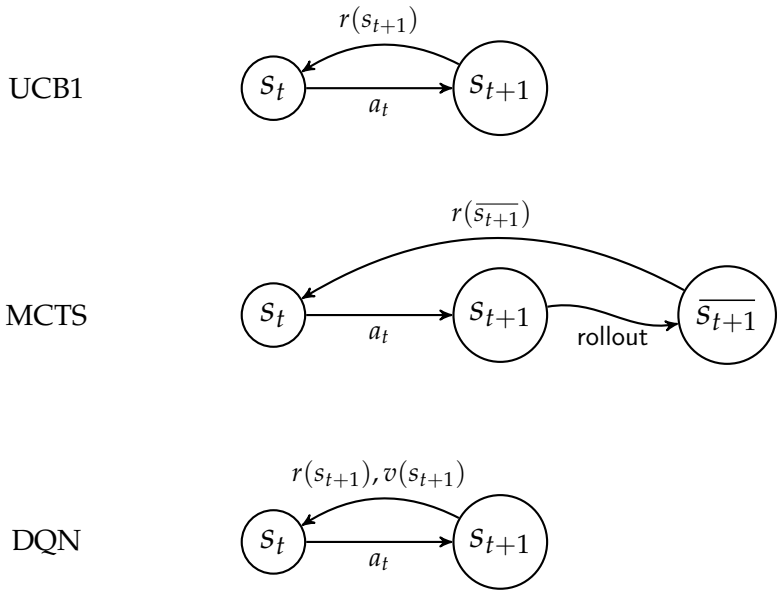


Figure 5.1: Different Reward Generation Techniques

5.1 Motivation

In this section, we want to explain why going towards an ordinal-based agent framework is an interesting path to discover. First, we do a quick recap of three algorithms (UCB1, MCTS, and DQL), discuss why they got popular. Based on that, we show why ordinal feedback is worth exploring and show the limitations of the introduced algorithms. We point out, that ordinal versions of UCB1, MCTS, and DQL can solve a set of problems, that their original implementations do not support: problems with natural ordinal feedback.

No Alternative?
Usually, those problems are tackled by assigning numerical values to ordinal rewards, like assigning a 0 for a losing move or a 1 to a winning move and a 0.5 to a draw. But using numeric rewards also introduces linear trade-offs between each triplet of rewards, like (win, lose) = (draw, draw) for this example. Those trade-offs are impossible to be disabled so that you need to define a numerical trade-off between *patient dead*, *patient cured* and *patient ill* instead of defining *patient dead* $<$ *patient ill* $<$ *patient cured*.

5.1.1 Choice of Algorithms

The areas of machine learning and artificial intelligence recently get high attention for different reasons: Processing big data and detecting interesting patterns is a major challenge in many businesses and can lead to better-informed decisions. Assistance systems like a driver-aid

system in cars can assist the user by providing crucial information or even take over hole systems. Autonomous driving and fully automated assembly lines are topics of research and our global society. All of this is mainly supported by modern hardware, big data centers, and computing clusters - none of that has ever been so easily accessible even for hobbyists. Looking at publications, it can be seen that several topics are enjoying much attention, and even low-threshold media like video platforms or blog posts are filled with related topics.

There is a hype about ML and AI, and several algorithms for different problems are getting developed and enhanced at a great pace, ranging from very baseline and broadly applicable to super-specialized and engineered.

5.1.2 Ordinal Feedback

Many of those algorithms mentioned above can be seen as agents interacting with an environment, with the agents performing actions and getting some feedback from the environment, influencing future action choices (see Figure 5.2). Most popular and successful algorithms require this feedback to be of a real-valued scale, to enable the use of gradients and other real-valued math frameworks. Looking at finance and other business decisions, it is common to have real-valued feedback, like profit and loss or distance measures. But there are lots of other environments that naturally do not provide real-valued feedback, like emotions or categorical values (like win, draw, loss). It is popular to transform categorical values to a numerical scale, like win$=1$, draw$=0.5$, and loss$=0$, and behave as those were the true rewards. But recent work has shown that emotions lie on ordinal scales and translating them into a numerical scale introduces a bias to the data [48]. It can easily be seen that once values lie on a numerical scale with a distance measure, one can derive information like: *Looking at average performance, [win, loss] is just as good as [draw, draw]*(since both will result in an average value of 0.5). This might make sense in this case but does not apply to the generic case or emotions. Just think of trading off *happy*, *sad*, and *bored* to achieve something like *calm*. The general preference-based motivation from Section 4.2 applies here, too.

The main point of going with ordinal instead of preference-based feedback is the (re)usability of single-arm reward signals. In Preference-Learning, a reward signal is the result of a comparison of two options. The only information gained here is which of both options performed better in a single trial. If that result originates from direct competition with a winner and a loser, preference-based feedback is a very suitable way to model that feedback. But the algorithms we focus on in this thesis have a different way to generate their feedback: Instead of competing in a paired match, each option is evaluated alone and their individual results are then compared to each other. This can be turned into a preference-based view by always picking two options

and creating a preference feedback based on the individual feedback. This idea has been followed in chapter 4, leading to the problem of symmetric and exponential growth and bad usage of data. These problems can be solved by delaying the comparison between options: Since options can be evaluated alone, one can also take the result of an option, store it and compare it with any past results of other options. This way, an evaluation does not only generate insight about a pair of options but insight about all options. Additionally, one is not forced to always choose two options to compare to each other. The fall-back solution in dueling bandits for this problem is that one can choose an arm twice, which will not generate any information at all. We call the method of storing rewards for later comparison *ordinal-based feedback* since we assume that each of those reward signals lies on an ordinal scale. This means, that one can derive preferences over two rewards but one does not need to care about any metric while generating rewards. Only the order is of interest here.

5.2 Ordinal Markov Decision Process

Markov Decision Process

Similar to the standard Markov Decision Process, [43] defines an ordinal version of an MDP with the only difference that the reward function R is to return ordinal rewards instead of numerical ones. Thus, it maps executing action a in state s and reaching state s' to an ordinal reward r_o from a subset of possible ordinal rewards O. In this chapter, for convenience, we assume that the set of ordinal rewards is modeled by a subset of natural numbers $O = \{o_1, ..., o_u\} \subset \mathbb{N}$, with u representing the number of ordinal rewards. W.l.o.g. we assume that the ordinal rewards are ordered by their index, such that $o_i < o_j \Leftrightarrow i < j$. Whereas a real-valued reward provides information about the qualitative size of the reward, the ordinal scale breaks rewards down to naturally ordered *reward tiers*. These reward tiers solely represent the rank of the desirability of a reward compared to all other possible rewards, which is noted as the ranking position i of a reward o_i in the set of all possible rewards $\{o_1, ..., o_u\}$ (compare Section 3.5). Interpreting the reward signals on an ordinal scale still allows us to order and directly compare individual reward signals, but while the numerical scale allows for comparison of rewards utilizing the magnitude of their difference, ordinal rewards do not provide this information. That is where the OAF helps out: using aggregation of ordinal rewards, we can construct a meaningful comparison of different actions.

5.3 The Ordinal Bandit Framework

We try to bridge a gap between non-numerical feedback environments and the great number of well-engineered real-value algorithms. As described at the beginning of this chapter, we try to achieve this

Algorithm 2 The OAF.Sample method

Input : The current state s_t, the past aggregation data D_t
Output : The new state s_{t+1}, the played action a_{t+1} and sampled reward r_{t+1}

$A_t \leftarrow Environment.PossibleActions(s_t) = A(s_t)$
$V_t \leftarrow Aggregator.GetRatings(A_t, D_t)$
$a_t \leftarrow Agent.ChooseAction(s_t, V_t, A_t)$
$\bar{r}_t, s_{t+1} \leftarrow Environment.ExecuteAction(s_t, a_t)$
return $[a_{t+1}, s_{t+1}, r_{t+1}]$

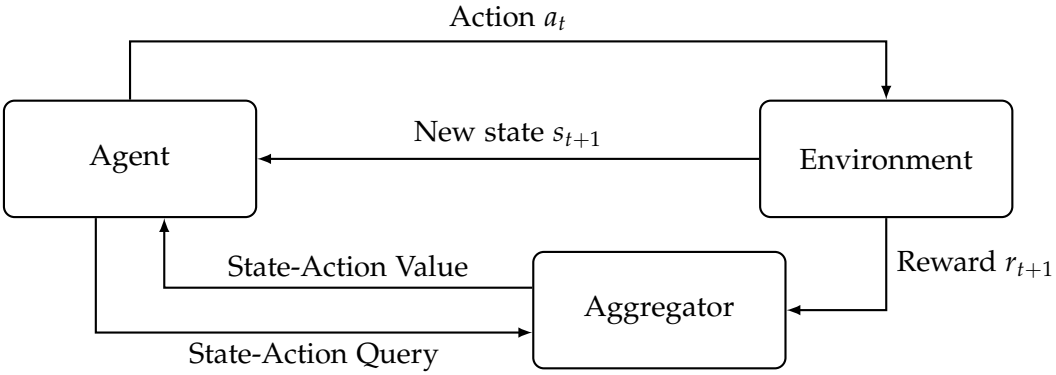


Figure 5.2: The framework setup

by formalizing and re-engineering an aggregation step which can be found in several algorithms that are based on *Markov Decision Processes* (MDPs). We introduce a method on how the aggregation step could be modified to enable the aggregation of ordinal data. In the following chapters, we see how this modification is applied to MAB, MCTS, and DQN. But first, we start with a formalization of the aggregation step.

Multi-Armed Bandit
Monte Carlo Tree Search
Deep Q-Network

5.3.1 The Aggregation Step

As described earlier in this chapter, we have identified a common concept in several algorithms that are based on an MDP: In an iterative manner, the agent executes one action and receives a reward. That reward is used to update an internal state that represents the quality of that action. We call that internal state an *Aggregation*. In upcoming iterations, the reward itself is not of interest, as the Aggregation provides all necessary information. In this section, we propose a formalization for this Aggregation step.

We do start by adding the aggregator as an actor to the common agent-environment framework known from Figure 2.1. The aggregator takes the rewards provided from the environment and supplies the agent with action-values (see Figure 5.2). An algorithm execution using this framework procedures as follows (compare Algorithm 2): In each time step t, the agent is in state s_t and the aggregator has some data d_t.

In this state, the agent can choose one action from the set of possible actions $a(s_t)$ defined by the environment. The aggregator rates each of those actions using its data, providing $V_t : A(s_t) \mapsto \mathbb{R}$ to the agent[1]. Now, the agent chooses the action to execute a_t based on the action values V_t. The environment executes this action a_t, returning a reward r_t and the next state s_{t+1}. Using the new reward r_t together with its transition data s_t, a_t, s_{t+1}, the aggregator can update its data, leading to d_{t+1}. This data d_{t+1} will then be used in the next iteration $t+1$ to rate the actions available in s_{t+1}. To summarize, the OAF consists of two methods that all algorithm implementations will share: The OAF.*Sample* method (see Algorithm 2) that chooses the next action to be played and the OAF.*Update* method that updated the aggregated data.

Ordinal Agent Framework

In the following chapters, we will introduce several algorithms using OAF. For those algorithms, we will provide pseudocode showing how the OAF.*Sample* and OAF.*Update* methods are used in detail. Those algorithms to state how the aggregation data D is set up (define which information is stored) and need to implement the following methods:

1. $V \leftarrow GetRatings(A', D)$ that assigns rating values for all actions $a \in A'$ using the aggregation data D.
2. $a \leftarrow ChooseAction(s_t, V_t, A'_t)$ chooses an action to play using the rating values provided by *GetRatings*.
3. $D \leftarrow \text{OAF}.Update(D, a, r)$ updates the aggregation data of action a with reward r.

As the first two methods are used in the OAF.*Sample* method, the latter will be directly used in the algorithm pseudocode together with OAF.*Sample*.

5.4 Conclusion

In this chapter, we have introduced a framework to separate environment rewards from learned action-values using an aggregator as an intermediate actor between the agent and the environment. This aggregator is in charge of processing the rewards received from the environment and supplying aggregated action-values to the agent on demand.

In the following chapters, we show how to implement UCB0, MCTS, and DQN following this framework. Each chapter contains a detailed evaluation of this framework comparing the ordinal with the original implementation. In Chapter 7 we also compare against PB-MCTS to check if our original assumption leading to the creation of this framework holds (removing the drawbacks of PB-MCTS to yield a better performance).

1 We limit ourselves to $\mathbb{R}$ since all inspected algorithms use real-valued feedback

6 Ordinal Multi-Armed Bandits

Starting with this chapter we introduce and implement algorithms using the Ordinal Agent Framework. Here, we start with the Multi-Armed Bandits setting showing how to implement the common UCB1 algorithm in our framework. Additionally, we introduce the Borda Score which is used by different algorithms in the upcoming chapters. Following the main research question, we transform UCB1 into an ordinal algorithm using the Borda Score. This leads to our first novel algorithm: Borda-UCB, which can use ordinal feedback instead of being reliant on numerical reward values. The following is based upon our publications [16, 17].

6.1 Introduction

MABs are a popular research topic and the origin of multiple popular algorithms used in real-world world systems. While not being too complex to understand in its idea, it is heavily researched especially when looking at its mathematical theory. A wide range of algorithms and problem definitions are based on MABs and are often found in economy and game theory: Their application ranges from ranking to element selection and they form a basis of other, more complex algorithms like MCTS, which internally relies on solving multiple MABs. The popularity and simplicity of MABs form a great basis to introduce our Bandit Framework: We are looking at UCB1 (see Section 2.4.4), discuss different possibilities on how it can be transformed into an ordinal algorithm, what kind of problems can be solved using it, implement it, and test our implementation comparing it to other algorithms on a synthetical data set.

In this chapter, we want to show how a numerical algorithm can be turned into a qualitative/ordinal algorithm on the example of UCB1. We introduce the Borda-Score, a technique to aggregate ordinal feedback into numerical values. Using that Borda-Score we turn UCB into Borda-UCB, revealing a novel algorithm solving the Qualitative Dueling Bandit problem. We analyze alternative approaches for that algorithm, introducing the underlying voting problem and pointing to alternatives. We further look at an alternative algorithm and compare it to our approach on a synthetical dataset.

Voters	Ranking
30	$A > B > C$
1	$A > C > B$
29	$B > A > C$
10	$B > C > A$
10	$C > A > B$
1	$C > B > A$

Table 6.1: 81 ranked votes among candidates A, B, and C

6.2 Voting

As introduced and noted in Section 2.3, we need to define *the optimal arm* a^* to measure the quality of bandit algorithms. When abstracting from numerical rewards to ordinal rewards, the idea of maximizing the cumulative reward is not applicable anymore as it is not possible to add or average ordinal rewards. Instead, we need another concept of optimality based on ordinal rewards. This is no new problem and there are different ways of defining the optimal arm, or, as it is often called in the literature when talking about ordinal feedback, the *winner*. *Voting theory* is a mathematical discipline centered around the concept of defining and identifying the winner of voting [6].

6.2.1 Identifying a Winner

In voting theory, a major question to answer is: *Who should be elected?* For this thesis, we look at ranking methods where a vote consists of a full ranking of all candidates. Having three candidates A, B, and C a valid vote could be $B > A > C$. In our case, when working with an OMDP, we do not receive direct preference information like that. Instead, when playing an arm, we receive an ordinal reward that can be stored and used for comparisons. Table 6.1 shows an outcome of a vote with three candidates and 81 votes [9]. Table 6.2 shows how those votes can be written as ordinal information with three arms (the candidates), three ordinal rewards and 81 reward samples for each arm aggregating the number of times a candidate received placing 1, 2, or 3. In the following, we introduce and discuss the Condorcet method and the Borda method, where the latter one is used in our upcoming algorithms.

6.2.2 Condorcet Method

The *Condorcet Winner* is one of the most often used winner definitions[49]. The origin motivation is that if one candidate beats every other candidate in direct comparison, this candidate is the *Condorcet*

	o_3	$< o_2$	$< o_1$
a_A	11	38	31
a_B	11	31	39
a_C	59	11	11

Table 6.2: 81 ranked votes displayed telling which candidate got which place how often.

Winner. In many cases, a Condorcet winner may not exist: Cyclic preferences like $A > B > C > A$ or ties like when $A > B$ and $B > A$ appear equally often can rule out the existence of a Condorcet winner. A voting method can fulfill the *Condorcet Criteria* telling that if a Condorcet winner exists, that method will designate the Condorcet winner as the winner of a vote. Many popular voting methods do not fulfill this criterion, like plurality voting or the Borda method introduced in the following chapter.

Looking at our application with OMDP returning ordinal rewards, we do not have direct access to preferences like from Table 6.1. Instead, we rely on an aggregated form as shown in Table 6.2. The Condorcet winner for this example is A, as it is $A > B$ in 41 and $A > C$ in 60 of 81 cases. In fact, the Borda method will not elect the Condorcet winner, but B as shown in the following section.

6.2.3 Borda Method

As alternatives to the Condorcet method described above, there are other winner election methods, one of which is the Borda method [5, 6]. In comparison to the Condorcet method where one needs to beat other arms in a pairwise comparison, the Borda method also includes a sense of by how far one wins or looses: We introduce the Borda rule, assigning a Borda count to each candidate. The candidates sharing the highest score are called Borda winner. With $\bar{c}$ being the number of candidates, the number of votes $\bar{v}$ and $0 < p_{i,j} \leq \bar{c}$ being the placement of candidate $0 \leq i < \bar{c}$ in the jth vote, we define the Borda count $BC(i)$ for candidate i as

$$BC(i) = \sum_{j}^{\bar{v}} \bar{c} - p_{i,j}. \tag{6.1}$$

Using the Borda method, we elect B as the winner with $BC(B) = 109$, $BC(A) = 101$, and $BC(C) = 33$.

In the following, we use the concept of the Borda Winner to identify the optimal arm of OMDPs. Looking further, we will also use the Borda Winner for our ordinal variations of MCTS and DQN (see Chapter 9 and Chapter 7).

6.3 Borda UCB

As introduced in the previous chapter, the Borda method does not rely on direct comparisons between competitors. Instead, it is based on an aggregation (see Table 6.2) comparing the distribution of ranks. Based on the Borda count, the Borda score estimates the mean performance of one competitor against the others or the average of the winning probabilities against other arms [39, 47]. Therefore, the Borda score of action a_1 depends on the alternatives and we denote it with $B_A(a_1)$, where A is the set of possible actions. In the case of O-MABs, $B_A(a_1)$ is the probability of action $a_1 \in A$ to win against any other action $a_2 \in A \setminus \{a_1\}$ available (with tie correction):

$$B_A(a_1) = \frac{1}{|A| - 1} \sum_{a_2 \in A \setminus \{a_1\}} \Pr(a_1 \succ a_2)$$

where

$$\Pr(a_1 \succ a_2) = \Pr(X_{a_1} > X_{a_2}) + 1/2 \Pr(X_{a_1} = X_{a_2})$$

and X_a is the random variable responsible for sampling the ordinal rewards for arm a. The true value for $\Pr(X_a > X_{a_2})$ and $\Pr(X_a = X_{a_2})$ is unknown for the algorithm but can be estimated empirically [41]: To do so, we introduce the empirical cumulative distribution function $\hat{F}_a(o)$ that measures the number of ordinal values $o' \leq o$ that have been sampled by playing the arm a. For simplicity, we also introduce the non-cumulative distribution function $\hat{f}_a(o)$ which measures the number of times that exactly o has been sampled. An estimation on $B_A(a)$ can be calculated given $\hat{F}_a(o)$ (we omit $\hat{f}_a(o)$ since it is given by $\hat{F}_a(o)$).

$$\begin{aligned} &\hat{B}_{A,F}(a_1 \succ a_2) \\ &= \int \Pr[X_{a_1} < o] \hat{f}_{a_2}(o) + 1/2 \Pr[X_{a_1} = o] \hat{f}_{a_2}(o) do \\ &= \sum_{i=2}^{|O|} \frac{F_{a_1}(o_{i-1}) + \hat{F}_{a_1}(o_i)}{2} \hat{f}_{a_2}(o_i) \end{aligned} \tag{6.2}$$

Using those pairwise Borda-Score estimates, one can estimate the Borda-Score $\hat{B}_{A,F}(a)$ of arm a by averaging over all pairwise Borda-Score estimates over all competitor arms in A:

$$\hat{B}_{A,F}(a) = \frac{1}{|A| - 1} \sum_{a_2 \in A \setminus \{a\}} \hat{B}_{A,F}(a \succ a_2) \tag{6.3}$$

Formula 6.3 defines the backbone of the following ordinal algorithms we present, as it manages to transform multiple ordinal rewards for different arms into one numerical value per arm. This numerical value $\hat{B}_{A,F}(a)$ is defined such that the expected arm maximizing this score is the Borda-Winner: $\arg\max_a E[\hat{B}_{A,F}(a)] = \arg\max_a B_A(a) = a^*$. The base idea we are going to follow is to use $\hat{B}_{A,F}(a)$ as the aggregated quality value for a and rely on well-performing numerical algorithms

to identify the Borda-Winner. This way we can transform algorithms that rely on numerical reward values to measure a numerical quality value and maximizing the cumulative reward into algorithms that identify the Borda-Winner for problems with ordinal rewards. In the following section, we show how the UCB1 algorithm can be modified this way to find the Borda-Winner. In Chapter 7 and Chapter 9 we follow a similar concept for MCTS and DQL.

Comparing Borda-UCB with UCB1, the former tries to play the action that beats the other actions most often whereas the latter maximizes the average performance. Looking at a comparison in terms of computation time, the biggest difference is that UCB1 calculates a running average of the rewards sampled per arm where Borda-UCB needs to calculate estimations of probabilities. As a running average can be calculated in a constant time, calculating the Borda-Score using Formula 6.3 iterates over all arms and ordinal reward values. The number of actions $|A|$ usually is relatively small compared to t or $|O|$. The number of received ordinal rewards can not be bound that easy, such that $|O|$ can linearly grow with t. Concluding, with UCB1 lying in the time-complexity class $\mathcal{O}(t)$ at time t, Borda-UCB (as calculated with Formula 6.3) lies in $\mathcal{O}(t|O||A|)$. In Chapter 8, we propose a method of improving the time complexity, such that the Borda-Score can be approximated without a time dependence on the number of ordinal rewards $|O|$.

We want to emphasize that another research group has developed the same concept of Borda-UCB as we did in parallel. While we developed Borda-UCB as the first step towards Borda-MCTS, [47] independently proposed an analogous algorithm and devoted substantial efforts to a thorough analysis of its theoretical background.

6.4 Using the Framework

In this section, we show how the UCB1 can be implemented using the agent framework. In the following section, we then derive Borda-UCB from it, enable the use of ordinal feedback, and identifying the Borda-Winner.

6.4.1 MAB

Ordinal Agent Framework

In this section, we show how the UCB1 algorithm (see Chapter 2.4.4) can be formulated using the Agent Framework. The OAF itself provides a scheme on how rewards are stored and used. To do so, one needs to provide implementations for different methods (like *Aggregator.Update*, compare Algorithm 2) and define how that framework instance is used. Algorithm 3 defines the latter, providing an overview of the algorithm. Basically, the OAF.*Sample* method chooses the arm to play and returns an update method and the sampled reward. As

Algorithm 3 The UCB1 Algorithm using the OAF

Input : The set of actions A and time horizon n and state s_0
Output : A candidate for the optimal action a^*

```
D ← empty aggregation data
while t < n do
  t ← t + 1
  a_t, _, r_t ← OAF_UCB1.Sample(D, s_0)
  D ← OAF_UCB1.Update(D, a_t, r_t)
end while
return action a played most often (see D)
```

by the design of UCB1, the sampled reward is directly used to update the action value.

In the following, we present implementations for the internal OAF methods.

$$UCB1 = \bar{r}_{a,t} + \sqrt{\frac{2 \ln n}{n_a}} \tag{6.4}$$

To do so, we first inspect what UCB1 aggregates per action. Looking at the UCB1 Formula 6.4, we see that we need to know three information: the average reward, the number of pulls per arm n_a and the sum of all pulls n. The latter can be calculated using $n = \sum_a n_a$, but the average reward $\bar{r}_{a,t}$ and number of pulls per arm n_a need to be stored. Hence, $(n_a, \bar{r}_{a,t})$ is the information stored internally per arm. Looking at our framework, the Aggregation Data D_t holds all information for each arm. For UCB1, this is $D_t = (n_a, \bar{X}_a)_{a \in A}$. Whenever we receive a new reward for an arm a, we need to update $(n_a, \bar{X}_a)$, such that n_a gets increased by 1 and $\bar{X}_a$ includes the new reward. The selection strategy of what arm to sample next can be carried over from Formula 6.4 using the data D_t. In the following, we summarize the implementation of UCB1 in the Agent Framework:

1. The Aggregator state at time t: $D_t = ((samples_{a,t}, value_{a,t}))_{a \in A}$ holds the information of how often an arm a has been played ($samples_{a,t}$) and the average reward of those samples ($value_{a,t}$). In short, we call $d_{a,t} := (samples_{a,t}, value_{a,t})$.

2. The Aggregator Update: $D_t, s_t, a_t, r_t, s_{t+1}$
   Update the information for action a_t to incorporate the new reward r_t

   $samples_{a_t,t+1} \leftarrow samples_{a_t,t} + 1$
   $value_{a_t,t+1} \leftarrow \frac{value_{a_t,t} \cdot samples_{a_t,t} + r_t}{samples_{a_t,t}+1}$
   The values for the other actions $a' \neq a_t$ are getting carried over from the last time step:
   $samples_{a',t+1} \leftarrow samples_{a',t}$
   $value_{a',t+1} \leftarrow value_{a',t}$

3. Get Ratings: (How to calculate ratings for the set of actions A given the Aggregator data D_t)
   $V_t \leftarrow A, D_t$
   $\forall a \in A : V_t(a) = d_{t,a}$

4. Agent.ChooseAction (How to choose the next action to play)
   $a_t = \arg\max_{a \in A} d_{a,t}^{(value)} + \sqrt{\frac{\ln d_{a,t}^{(samples)}}{\sum_{a' \in A} d_{a',t}^{(samples)}}}$

6.4.2 OMAB

Following the Borda UCB concept presented in Section 6.3, we now show how it can be implemented in our Agent Framework. Thereby, we focus on showing the similarities and differences compared to the MAB implementation from the previous section.

The base idea of Borda-UCB is to approximate the reward distributions F_a for each arm a. Given those approximate distributions $\hat{F}_a$, the pairwise winning probabilities $\Pr(a_i \succ a_j)$ and Borda scores $\hat{B}$ can be approximated (see Formula 6.3). Hence, we need to change the Aggregator Data D_t and the GetRatings method when looking at implementation differences from Borda-UCB to UCB1: To be able to use Formula 6.3 from Section 6.3 we store the sampling count of receiving reward $o \in O$ for each action $a \in A$: $\hat{f}_a(o)$ in the aggregation data. Looking at the GetRatings function $V_t \leftarrow A, D_t$ we now can calculate the estimated Borda Score $\hat{B}_{A,a}()$ for each action $a \in A$ and use that as the rating for all action $\forall a \in A : V_t(a) = \hat{B}_{A,a}()$.

With those modifications in place, we have turned UCB1 into Borda-UCB, since we now choose the arm that maximizes the Borda-Score.

Borda-UCB ([**Borda-UCB**]) introduces some further algorithmically changes: In Borda-UCB the Borda Score is used to identify and exploit good performing arms. A common approach to also achieve exploration in bandit algorithms is to use an estimation on the upper bound of that exploitation term as a selection strategy. The upper bound is high for both, well performing arms as well as for arms with a low sample size while smoothly interpolating between those two extremes:

$$\textit{Borda-UCB}_{A,\hat{F}}(a) = \hat{B}_{A,\hat{F}}(a) + \left(\sqrt{\frac{\alpha \log n}{n_a}} + \frac{1}{|A| - 1} \sum_{b \in A \setminus \{a\}} \sqrt{\frac{\alpha \log n}{n_b}} \right), \tag{6.5}$$

where $\alpha > 0$ is a control parameter to trade-off exploration and exploitation. Notice, that the posterior summand (the exploration-term) for action a is not only dependent on the relative number of samples of a, but also depends on the complete sample distribution over all arms. Since the Borda-Score estimate $\hat{B}_{A,F}(a)$ is averaged over multiple pairwise arm comparisons it is necessary to have a good estimate for each pair to receive a tight upper bound for the average.

Additionally, the selection strategy of Borda-UCB is slightly different from common state of the art algorithms, too. A common strategy

used is to pick the arm that maximizes the estimated upper bound and sample a value for it. Contrary to that, Borda-UCB separates the arms into two sets: The exploitation and exploration set, where the first is composed of all arms having the currently best exploration values and the latter being the other arms. In each time step t we call $a_t = \arg\max \textit{Borda-UCB}_A(a)$ the selected arm, which is played and updated. Instead of moving to the next step, we check whether a_t is an element of the exploration set. If so, the complete exploration set is played sequentially. This way it can be assured that each arm is played regularly and thus the upper bound is not becoming too loose. For a more formal derivation, we refer to [47].

6.5 Related Work

A related topic to ordinal bandits is dueling bandits, where at each time step two arms are pulled and the agent gets one reward indicating which arm won in the direct comparison (see Chapter 3.4). In reality, repeated dueling of different pairs is often used to identify the best element of a set, like in most sports leagues where different teams battle each other. The biggest downside of this approach (at least from an energy or optimization point of view) is that each team needs to play against each other team at least once to be able to identify a winner. If it would be possible to measure teams independently, only one measure per team would suffice to identify the winner. Sample efficiency is the biggest difference between dueling and ordinal bandits: In ordinal bandits, it is possible to measure the quality of one action, where for dueling bandits it always needs two actions to be compared against each other. But not every problem definition can be solved by individual measures of arms (like soccer).

It has been shown that it is possible to reduce dueling bandits to common numerical bandits[1]. Hence, it is possible to optimize a bandit using preference information by relying on numerical bandit algorithms. The idea here is to create artificial numerical bandits for each pair of actions to predict pairwise comparisons. This way it is possible to reduce pairwise comparison to numerical comparisons, but the sample efficiency stays the same as for dueling bandits. MultiSBM is an algorithm of that kind as will be used in the following experiments to benchmark against our Borda-UCB algorithm.

The QUCB algorithm (Qualitative-UCB) takes another route: Instead of relying on pairwise preferences, *Qualitative Upper Confidence Bound* (QUCB) can directly be applied on numerical or ordinal MDPs and has an additional threshold parameter $\tau \in [0, 1]$. This parameter τ defines a representative reward for a set of sampled rewards: the τ-percentile, or the best reward in the worst $\tau \cdot 100\%$ of sampled rewards. Just like for Borda-UCB, QUCB stores a cumulative distribution function over the sampled rewards for each arm. At every time step t, the τ-percentile of arm a $\mathrm{T}_{a,t}$ can be calculated using the CDF. Now each arm has its representative reward $\mathrm{T}_{a,t}$ and the max arm can be

Problem	Arm1	Arm2	Arm3	Arm4	Arm5	Arm6
Bernoulli	0.8	0.7	0.7	0.2	0.2	0.2
Log	0.9	0.83	0.74	0.64	0.51	0.32
Exp	0.9	0.33	0.12	0.05	0.02	0.01

Table 6.3: Rewards and probabilities

sampled. But instead of directly using the τ-percentile, each arm adds an exploration bias to τ, such that rarely sampled arms may use its 0.7-percentile where an often sampled arm uses its 0.5-percentile as its representative $T_{a,t}$. The performance of QUCB is reliant on a good choice of parameter τ.

6.6 Experiments

The experiments in this chapter are to show that the base idea of Borda-UCB works and that it has a better sample efficiency than the pairwise approach as used by MultiSBM. The experiments are based on synthetic data to highlight differences between UCB1 and Borda-UCB as well as a better convergence speed of Borda-UCB compared to the dueling approach of MultiSBM.

In our experiments, we test our novel Borda-UCB against MultiSBM and UCB1 on different problems. Each arm is generating a numerical reward to be used by UCB1. Borda-UCB and MultiSBM are using ordinal rewards derived from the numerical rewards. Even though the regret definitions of the three algorithms do not match, they all try to identify the optimal arm and play it most often. Hence, we look at two values when analyzing the performance of an algorithm: The numerical regret as optimized by UCB1 and the accuracy of playing the optimal arm a^* (which in most cases is the same arm for the algorithms). Each problem is repeated 32000 times and the resulting experiments are averaged. Since Borda-UCB comes with an exploration-exploitation parameter a but UCB1 does not, we fix $a = 1$ for the experiments.

6.6.1 Used Bandits

The first bandit (called *One-Best*) is used to compare all three algorithms in terms of convergence. Here, the algorithms can choose between six arms with one optimal arm, two close-to-optimal, and three bad choices. We analyze how fast each algorithm is able to identify the optimal arm, especially checking if Borda-UCB is able to outperform MultiSBM. Each arm samples from a Bernoulli distribution, sampling either a 0 or 1 depending on the given probabilities shown in Table 6.3.

Problem	Arm1	Arm2
BanditA	75% : 1 25% : 0	100% : 0.5
BanditB	75% : 0.6 25% : 0	100% : 0.5

Table 6.4: Winning probabilities on *BanditA* and *BanditB* per arm. Looking at *BanditA*, Arm1 wins in average and by direct comparison against Arm2. At *BanditB*, Arm1 also wins in direct comparison, but looses on average against Arm2.

		Arm1	Arm2	Arm3	Arm4	Arm5	Arm6
Log	Borda-UCB	31666	219	60	28	16	11
	UCB1	30125	1323	326	134	63	29
Exp	Borda-UCB	31666	219	60	28	16	11
	UCB1	31912	31	17	14	13	13

Table 6.5: Results of the *LogDistributed* and *ExpDistributed* problems. Which arm gets pulled how often over 32000 pulls.

This table also shows the rewards for two other bandits: *LogDistributed* and *ExpDistributed* used in the second experiment. The rewards for these two bandits are deterministic, returning the same reward every time an arm gets sampled. The main idea here is that the underlying numerical rewards differ while the ordering of the arms stays the same. From an ordinal perspective, the two bandits to not differ while UCB1 will have a harder time identifying the optimal arm for *LogDistributed* than for *ExpDistributed*.

To highlight that UCB1 and Borda-UCB do differ in terms of their optimization target, we run the third experiment and compare both algorithms on *BanditA* and *BanditB* with rewards shown in Table 6.4. By changing the reward distribution of the first arm (Arm1), UCB1 changes the definition of the best arm, where Borda-UCB does not change the behavior because the order of rewards does not change.

BanditA	Arm1	Arm2
Borda-UCB	31960	40
UCB1	31857	143

BanditB	Arm1	Arm2
Borda-UCB	31961	39
UCB1	2183	29817

Table 6.6: Results of the Borda-UCB vs. UCB1 experiment: UCB1 switches the arm by its average performance where Borda-UCB is only dependent on the ordering

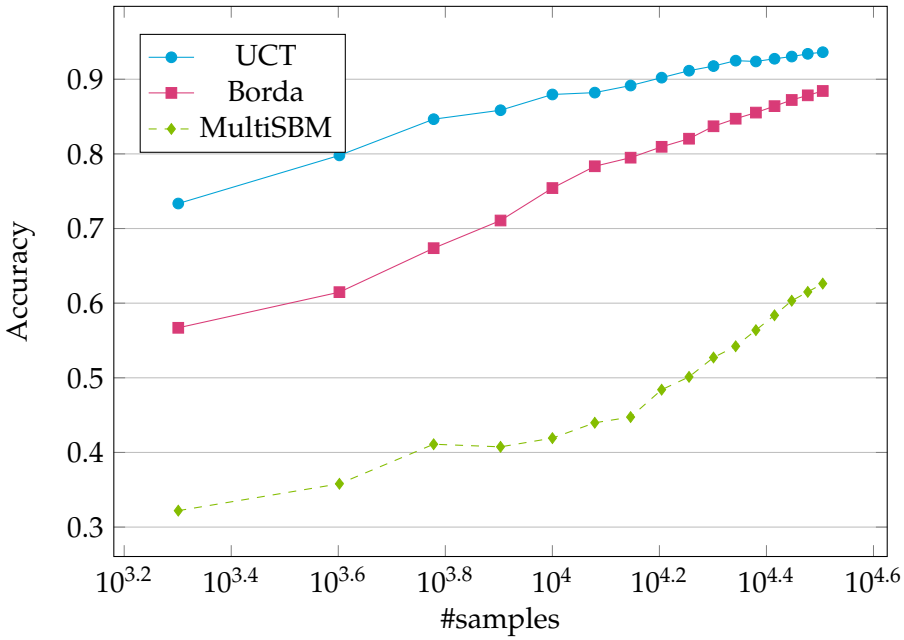


Figure 6.1: Results on how all three algorithms compare.

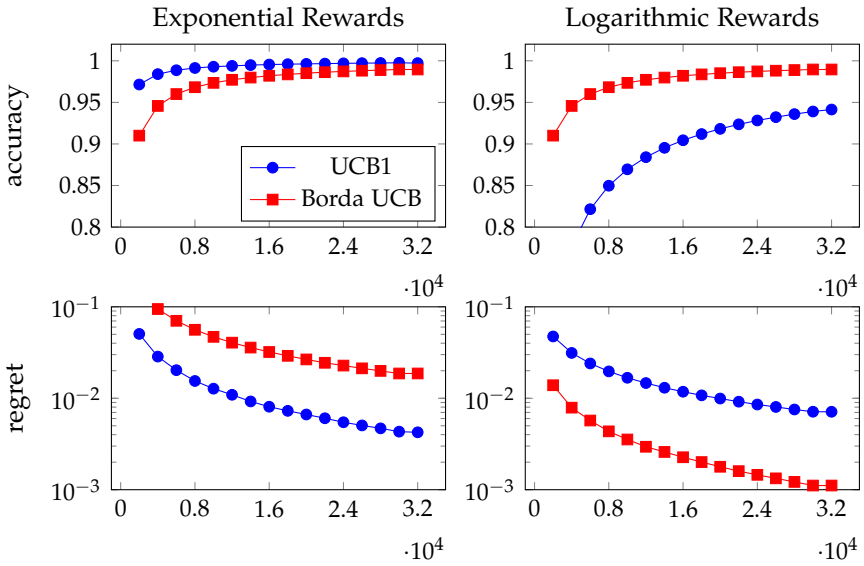


Figure 6.2: Results on the four experiments showing regret and accuracy.

We start the result discussions by looking at the *One-Best* problem. Here, the three algorithms UCB1, Borda-UCB, and MultiSBM are separately tested on a bandit where the arms have different probabilities of returning a 1 or a 0. In Figure 6.1, the results of these experiments are shown, plotting the accuracy of picking the optimal arm (Arm1) at each timestamp from 0 to 32000 plays. It can be seen, that UCB1 performs best with Borda-UCB being a close runner-up. MultiSBM performs worse, falling behind early without showing convergence within the time horizon of 32000 pulls. This shows that Borda-UCB is

able to utilize the rewards better than MultiSBM, as it can identify the optimal arm faster.

In the second experiment, we show that Borda-UCB is not dependent on a distance measure between rewards, compared to UCB1. Table 6.5 shows that Borda-UCB does not perceive a difference between those bandits while UCB1 is sensitive to the difference: Borda-UCB pulls the optimal arm in both cases 31666 times, where UCB1 pulls it 30125 times (for logarithmic rewards) and 31912 times (for exponential rewards). Figure 6.2 shows the numerical regret and accuracy of playing the optimal arm for both bandits and algorithms. As UCB1 performs better for *LogDistributed* with higher accuracy on playing the optimal arm and a lower average regret, Borda-UCB performs better *LogDistributed*. This shows that Borda-UCB can outperform UCB if the numerical rewards are not designed well.

The result of the last experiment highlights that Borda-UCB and UCB1 can converge to different optima. Since Borda-UCB favors the arm that beats the alternatives, UCB1 optimizes the average reward directly. As shown in Table 6.6, both algorithms favor arm a_1 in the first experiment, but they differ in the second experiment. Since in both cases, a_1 beats a_2 in 75% of pulls, Borda-UCB favors a_1 in both settings. Looking at the average reward, a_1 has an average reward of 0.75 in the first and 0.45 in the second experiment. With a_2 returning a reward of 0.5 for both bandits, UCB1 favors a_1 in the first but a_2 for the second bandit.

6.7 Conclusion

We have shown that it's possible to derive a qualitative Bandit algorithm from UCB1 using our bandit framework by utilizing the Borda Score. Instead of maximizing the cumulative reward, we hereby try to identify the Borda-Winner. Other algorithms are tackling the same problem definition (like MultiSBM). Experiments have shown that Borda-UCB is able to perform better than preference-based approaches due to a better sample efficiency. We also demonstrate that Borda-UCB and UCB1 differ, that they can have different optimal arms, and that Borda-UCB can outperform UCB1 in terms of numerical regret if the numerical rewards are poorly tuned.

7 Ordinal MCTS

In the past chapters, we have introduced PB-MCTS; an MCTS algorithm that can use preference feedback. The main disadvantage of this approach is an inefficient use of rewards by relying on preferences only and a restraint on the asymmetric growth of the search tree. Also, we have introduced Borda-UCB in the previous chapter: A MAB algorithm able to identify the Borda Winner using ordinal feedback. In this chapter, we leverage Borda-UCB to a Monte Carlo Tree Search algorithm and by this create a competitor to PB-MCTS. We introduce the General Video Game AI (GVGAI), a problem setting where agents need to play unknown games and show that our novel Borda-MCTS algorithm is able to beat vanilla MCTS that relies on numerical feedback. The following is based upon and in parts taken from our publication [16].

Monte Carlo Tree Search

Upper Confidence Bound

7.1 Introduction

Monte Carlo Tree Search (MCTS) is a popular algorithm to solve MDPs. MCTS is used in many successful AI systems, such as AlphaGo [36] or top-ranked algorithms in the general video game playing competitions [18, 30]. A reoccurring problem of MCTS with limited time resources is its behavior in case of danger:

Markov Decision Process

In this chapter, we present an ordinal MCTS algorithm that can utilize such ordinal feedback. We call our novel algorithm Ordinal MCTS (Borda-MCTS) and compare it to different MCTS variants on synthetic data that reflects our motivation as well as using the General Video Game AI (GVGAI) framework [32]. The algorithm is based on a novel ordinal bandit algorithm and leverages it into the area of tree search. Other than existing methods, we completely rely on ordinal information and do not translate those values into a numerical framework.

This work can be seen as a leverage of Borda-UCB to Borda-MCTS just like MCTS is a tree version of UCB1

7.2 The Borda-MCTS Algorithm

Here, we introduce Borda-MCTS as a variation of MCTS that allows for ordinal rewards. To do so, we first introduce how MCTS can be implemented in our framework. Then, we introduce Borda-MCTS showing the implementation differences to MCTS and analyzing the novel algorithm. In the next section, we then evaluate those two algo-

Algorithm 4 The MCTS Algorithm using the OAF

Input : The start state s_0
Output : A candidate for the optimal action a^*

```
D ← empty aggregation data
while computation time left do
  i = 0
  s̄_i = s_0
  while true do
    a_i, s̄_i, _ ← OAF_MCTS.Sample(D, s̄_i)
    if s̄_i is terminal then
      break
    end if
    if s̄_i is not yet in the search tree then
      Add s̄_i to the search tree
      break
    end if
  end while
  r ← Rollout(s̄_i)
  for all 0 ≤ î ≤ i do
    D ← OAF_MCTS.Update(D, a_î, r)
  end for
end while
return action a ∈ A(s_0) played most often (see D)
```

rithms together with PB-MCTS and some domain-specific algorithms using the General Video Game Framework.

7.2.1 MCTS

Multi-Armed Bandit

In this section, we show how the common MCTS algorithm can be implemented using our novel Agent Framework. Following the same concept as for the MAB algorithms, we motivate and show the implementation of vanilla MCTS (compare Section 2.5) using the Agent Framework interface. Pseudocode is given in Algorithm 4 with the method implementations given below. Without loss of generality, we assume for this implementation that different states do not share actions.

1. The Aggregator state at time t needs to store all the required information needed for the tree policy to function. For MCTS, this is the number of visits per action and their average value. Hence, that two pieces of information are stored in the aggregator state D. $D_t = ((samples_{a,t}, value_{a,t}))_{a \in A}$ holds the information of how often each arm has been played ($samples_{a,t}$) and the average reward of those samples ($value_{a,t}$). In short, we call $d_{a,t} := (samples_{a,t}, value_{a,t})$ the current information of arm a at time t.

2. The Aggregator Update: $D_t, s_t, a_t, r_t, s_{t+1}$ updates the aggregator state in the backpropagation phase.
   Update the information for action a_t and its parents $parents(a_t)$ to incorporate the new reward r_t $\forall \bar{a} \in parents(a_t) \cap \{a_t\}$:
   $samples_{\bar{a},t+1} \leftarrow samples_{\bar{a},t} + 1$
   $value_{\bar{a},t+1} \leftarrow \frac{value_{\bar{a},t} \cdot samples_{\bar{a},t} + r_t}{samples_{\bar{a},t}+1}$
   The values for all other actions not handled above are getting carried over from the last time step:
   $samples_{a,t+1} \leftarrow samples_{a,t}$
   $value_{a,t+1} \leftarrow value_{a,t}$

3. Get Ratings: (How to calculate ratings for the set of actions A given the Aggregator data D_t)
   $V_t \leftarrow A, D_t$
   $\forall a \in A : V_t(a) = d_{a,t}$

4. Agent.ChooseAction (How to choose the next action to play)
   $a_t = \arg\max_{a \in A} d_{a,t}^{(value)} + c \cdot \sqrt{\frac{\ln d_{a,t}^{(samples)}}{\sum_{a' \in A} d_{a',t}^{(samples)}}}$

7.2.2 Borda-MCTS

Monte Carlo Tree Search

In this section, we introduce Borda-MCTS, a novel Borda variation of MCTS that is able to process ordinal rewards and identify the Borda-Winner. Following the idea of the previous chapter, we will modify the Aggregation Step by incorporating the Borda Score instead of averaging numerical rewards. Hence, it will use the Borda-UCB instead of UCB as its tree policy.

Ordinal Monte Carlo tree search (Borda-MCTS) proceeds like conventional MCTS as introduced in Section 7.2.1, but replaces the average value $\bar{X}(a)$ in Formula 2.4 with the estimated Borda score $\hat{B}(a)$ used and introduced in the Borda-UCB algorithm (Section 6.3). Here, each arm is rated according to its mean performance against the other arms. Since there are no further changes in the algorithm, the pseudocode for MCTS (see Algorithm 4) also is valid for the *Borda Monte Carlo Tree Search* (B-MCTS) implementation and only method implementations differ. To our knowledge, Borda score has not been used in MCTS before, even though several papers have investigated its use in dueling bandit algorithms [15, 34, 39]. To calculate the Borda score for each action in a node, Borda-MCTS stores the backpropagated ordinal values, and estimates pairwise preference probabilities $\hat{B}(a \succ b)$ from these data. Hence, it is not necessary to do multiple rollouts in the same iteration as in PB-MCTS because current rollouts can be directly compared to previously observed ones.

In comparison to UCB, Borda-UCB uses the Borda score $\hat{B}$ as an exploitation term to choose the arm to play:

$$O\text{-}UCB = a^* = \arg\max_{a \in A} \hat{B}_A(a) + 2C\sqrt{\frac{2\ln n}{n(a)}}. \tag{7.1}$$

Note that $\hat{B}(a)$ can only be estimated if each action was visited at least once. Hence, similar to other MCTS variants, we enforce this by always selecting non-visited actions in a node first. Additionally, we did not use the exploitation term or selection strategy used in Borda-UCB, but carryover the common UCT exploration term and selection strategy. We motivate those choices in the following sections.

Exploration

The Borda-UCB exploration part for action a is composed of two parts: the uncertainty for action a itself and the averaged uncertainty of the other arms (compare Figure 6.5). Using $C = 2\frac{|A|-2}{|A|-1}\sqrt{\frac{\alpha}{2}}$ we can show that

$$\begin{aligned}
&\sqrt{\frac{\alpha \log n}{n(a)}} + \frac{1}{|A|-1} \sum_{b \in A \setminus \{a\}} \sqrt{\frac{\alpha \log n}{n(b)}} \\
&= \frac{|A|-2}{|A|-1}\sqrt{\frac{\alpha \log n}{n(a)}} + \frac{1}{|A|-1}\sqrt{\frac{\alpha \log n}{n(a)}} + \frac{1}{|A|-1} \sum_{b \in A \setminus \{a\}} \sqrt{\frac{\alpha \log n}{n(b)}} \\
&= \frac{|A|-2}{|A|-1}\sqrt{\frac{\alpha \log n}{n(a)}} + \frac{1}{|A|-1} \sum_{b \in A} \sqrt{\frac{\alpha \log n}{n(b)}} \\
&= \frac{|A|-2}{|A|-1}\sqrt{\frac{\alpha \log n}{n(a)}} + R \\
&= 2C\sqrt{\frac{2 \log n}{n(a)}} + R
\end{aligned} \tag{7.2}$$

with $R = \frac{1}{|A|-1} \sum_{b \in A} \sqrt{\frac{\alpha \log n}{n(b)}}$ being a constant term for all actions a per time step t. Inspecting Formula 7.2 and considering that we only want to identify the action that maximizes $\textit{Borda-UCB}(a)$, we can ignore R since it is the same for all actions. Thus, we can rewrite the exploration part of *Borda-UCB* to match Formula 7.1.

Tree Policy

As explained in Section 6.3, the selection policy of Borda-UCB distinguishes between playing an exploitation action and an exploration action. Whenever an exploration action is chosen, Borda-UCB evaluates all exploration actions at once to ensure a tight upper confidence bound[47]. This can not easily be done in a tree search setting: Evaluating multiple actions per node in a tree can cause an exponential

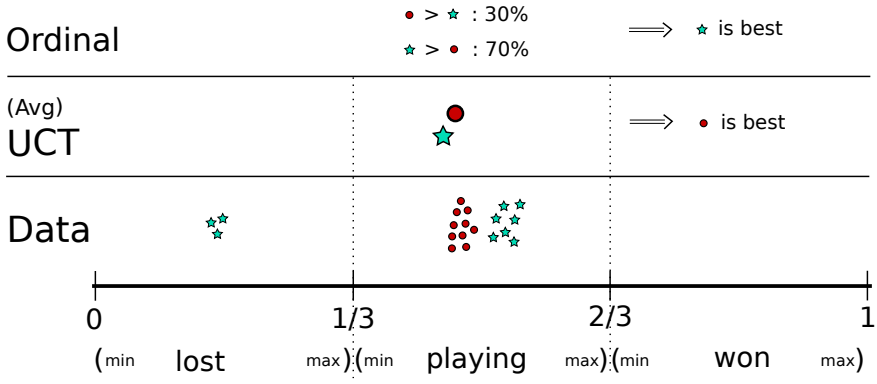


Figure 7.1: Two actions with different distributions.

number of paths explored since this split can happen at each level. The same problem occurred in PB-MCTS and led to performance issues. That is why we chose not to play all exploration actions and only evaluate the one action maximizing Formula 7.1.

Discussion

Although the changes from MCTS to Borda-MCTS are comparably small, the algorithms have very different characteristics. In this section, we highlight some differences between Borda-MCTS and MCTS.

Different Bias As mentioned previously, MCTS has been blamed for behaving cowardly, by preferring safe but unyielding actions over actions that have some risk but will in the long run result in higher rewards. As an example, consider Figure 7.1, which shows in its bottom row the distribution of trajectory values for two actions over a range of possible rewards. One action (circles) has a mediocre quality with low deviation, whereas the other (stars) is sometimes worse but often better than the first one. Since MCTS prioritizes the stars only if the average is above the average of circles, MCTS would often choose the safe, mediocre action. In the literature one can find many ideas to tackle this problem, like MixMax backups [21] or adding domain knowledge (e.g., by giving a direct bonus to actions that should be executed [18, 30]). Borda-MCTS takes a different point of view, by not comparing average values but by comparing how often stars are the better option than circles and vice versa. As a result, it would prefer the star action, which is preferable in 70% of the games. Please note that the given example can be inverted such that MCTS takes the right choice instead of Borda-MCTS.

Hyperparameters and Reward Shaping When trying to solve a problem with MCTS (and other algorithms, too), rewards can be seen as hyperparameters that can be tuned manually to make an algorithm work as desired. In theory, this can be beneficial since you can tweak the algorithm with many parameters. In practice, it can be very painful since there often is an overwhelming number of hyperparameters to

tune this way. This tuning process is called *reward shaping*. In theory, one can shape the state rewards until a greedy search can perform optimally on any problem.

Borda-MCTS reduces the number of hyperparameters by only asking for ordinal rewards; which is like asking for a ranking of states instead of real numbers for each state. This limits the possibilities of reward shaping but induces a fixed bias using the Borda method.

Computational Complexity Clearly, a naïve computation of $\hat{B}$ is computationally more expensive than MCTS' calculation of a running average. We hence want to point out that once a new ordinal reward is seen it is possible to incrementally update the current value of $\hat{B}$ instead of calculating it again from scratch. In our experiments, updating the Borda score needed 3 to 20 times more time than updating the average (depending on the size of O and A). These values do only show the difference in updating $\hat{B}$ in comparison to updating the running average, not the complete algorithms (where the factor is much lower, mostly depending on the runtime of the forward model).

7.3 Experiments

In our experiments, we compare our novel Borda-MCTS algorithm with PB-MCTS and vanilla MCTS. Since MCTS is a numerical algorithm, we introduce numerical and ordinal rewards for each setting. To preserve comparability, the ordinal rewards are derived from the numerical rewards by neglecting distances between numerical rewards.

7.3.1 Experimental Setup

We test the three algorithms described above (MCTS, Borda-MCTS, and PB-MCTS) using the General Video Game AI (GVGAI) framework [32]. As additional benchmarks, we add MixMax (Q parameter set to 0.25) as an MCTS variation that was suggested by [21] to tackle the cowardly behavior, and Yolobot, a state of the art GVGAI agent [18, 30]. GVGAI has implemented a variety of different video games and provides playing agents with a unified interface to simulate moves using a forward model. Using this forward model is expensive so that simulations take a lot of time. We use the number of calls to this forward model as a computational budget. In comparison to using the real computation time, it is independent of specific hardware, algorithm implementations, and side effects such as logging data.

Our algorithms are given access to the following pieces of information provided by the framework:

available actions: The actions the agent can perform in a given state

game score: The score of the given state $\in \mathbb{N}$. Depending on the game this ranges from 0 to 1 or -1000 to 10000.

game result: The result of the game: *won*, *lost*, or *running*.

simulate action: The forward model. It is stochastic, e.g., for enemy moves or random object spawns.

Heuristic Monte Carlo Tree Search

The games in GVGAI have a large search space with 5 actions and up to 2000 turns. Using vanilla MCTS, one rollout may use a substantial amount of time, since up to 2000 moves have to be made to reach a terminal state. To achieve a good estimate, many rollouts have to be simulated. Hence, it is common to stop rollouts early at non-terminal states, using a heuristic to estimate the value of these states. In our experiments, we use this variation of MCTS, adding the maximal length for rollouts *RL* as an additional parameter. The heuristic value at non-terminal nodes is computed in the same way as the terminal reward (i.e., it essentially corresponds to the score at this state of the game).

Mapping Rewards to $\mathbb{R}$

The objective function has two dimensions: on the one hand, the agent needs to win the game by achieving a certain goal, on the other hand, the agent also needs to maximize its score. Winning is more important than getting higher scores.

Since MCTS needs its rewards being $\in \mathbb{R}$ or even better $\in [0,1]$, the two-dimensional target function needs to be mapped to one dimension, in our case for comparison and ease of tuning parameters into $[0,1]$. Knowing the possible scores of a game, the score can be normalized by $r_{norm} = (r - r_{min})/(r_{max} - r_{min})$ with r_{max} and r_{min} being the highest and lowest possible score.

For modeling the relation $lost \prec playing \prec won$ which must hold for all states, we split the interval $[0,1]$ into three equal parts (cf. also the axis of Figure 7.1):

$$r_{mcts} = \frac{r_{norm}}{3} + \begin{cases} 0, & \text{if } lost \\ \frac{1}{3}, & \text{if } playing \\ \frac{2}{3}, & \text{if } won. \end{cases} \tag{7.3}$$

This is only one of many possibilities to map the rewards to $[0,1]$, but it is an obvious and straightforward approach. Naturally, the results for the MCTS techniques, which use this reward, will change when a different reward mapping is used, and their results can probably be improved by shaping the reward. In fact, one of the main points of our work is to show that for Borda-MCTS (as well as for PB-MCTS) no such reward shaping is necessary because these algorithms do not rely on the numerical information. In fact, for them, the mapped linear function with $a \succ b \Leftrightarrow r_{mcts}(a) > r_{mcts}(b)$ is equivalent to the preferences induced by the two-dimensional feedback.

Selected Games

GVGAI provides users with many games. Evaluating all of them is not feasible. Furthermore, some results would exhibit erratic behavior, since the tested algorithms (except for YOLOBOT) are not suitable for solving some of the games. For example, true rewards often are very sparse, and the agent has to be guided in some way to reliably solve the game.

For this reason, we manually played all the games and selected a variety of interesting, and not too complex games with different characteristics, which we believed to be solvable for the tested algorithms (see Table 7.1)

The number of iterations that can be performed by the algorithms depends on the computational budget of calls to the forward model. We tested the algorithms with 250, 500, 1000, and 10000 forward model uses (what is later called 'budget'). Thus, in total, we experimented with 28 problem settings (7 domains $\times$ 4 different budgets).

Parameter Tuning and Experiments

All MCTS algorithms have two parameters in common, the *exploration trade-off* C and *rollout length* RL. For RL we tested 4 different values: $5, 10, 25$, and 50, and for C we tested 9 values from 0 to 2 in steps of size 0.25. In total, there are 36 configurations per algorithm. To reduce variance, we have repeated each experiment 40 times. Overall, 4 algorithms with 36 configurations were run 40 times on 28 problems, resulting in 161280 games played for tuning.

Additionally, we compare the algorithms to YOLOBOT, a highly competitive GVGAI agent that won several challenges [18, 30]. YOLOBOT is able to solve games none of the other five algorithms can solve. Note that YOLOBOT is designed and tuned to act within a 20ms time limit. Scaling and even increasing the budget might lead to worse and unexpected behavior. Still, it is added for sake of comparison and interpretability of strength. For YOLOBOT each of the 28 problems is played 40 times, which leads to 1120 additional games or 162400 games in total.

We are mainly interested in how well the different algorithms perform on the problems, given optimal tuning per problem. To answer, we show the performance of the algorithms per problem in the percentage of wins and obtained average score. We do a Friedmann test on average ranks of those data with a posthoc Wilcoxon signed-rank test to test for significance [10]. Additionally, we show and discuss the performance of all parameter configurations.

7.3.2 Experiment Results

Table 7.2 shows the best win rate and the corresponding average score of each algorithm averaged over 40 runs for each of the 36 different parameter settings. In each row, the best values for the win rate and

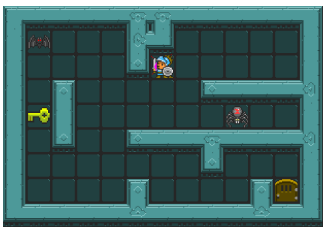
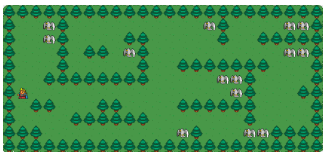
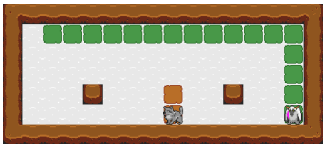
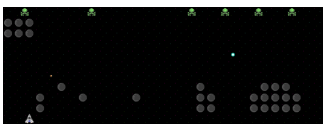
Game	Description
Zelda 	The agent can hunt monsters and slay them with its sword. It wins by finding the key and taking the door.
Chase 	The agent has to catch all animals which flee from the agent. Once an animal finds a caught one, it gets angry and chases the agent. If the agent gets caught this way, the game is lost.
Whackamole 	The agent can collect mushrooms that spawn randomly. A cat helps it in doing so. The game is won after 2000 time steps or lost if the agent and the cat collide.
Boulderchase 	The agent can dig through sand to a door that opens after it has collected ten diamonds. Monsters chase it through the sand turning sand into diamonds. It may be very hard for an MCTS agent to solve this game.
Surround 	The agent can win the game at any time by taking a specific action or collect points by moving while leaving a snake-like trail. A moving enemy also leaves a trail. The agent shall not collide with trails.
Jaws 	The agent controls a submarine, which is hunted by a shark. It can shoot fish giving points and leaving an item behind. Once 20 items are collected, a collision with the shark gives a large number of points, otherwise it loses the game.
Aliens 	The agent can only move from left to right and shoot upwards. Aliens come flying from top to bottom throwing rocks on the agent. For increasing the score, the agent can shoot the aliens or shoot disappearing blocks.

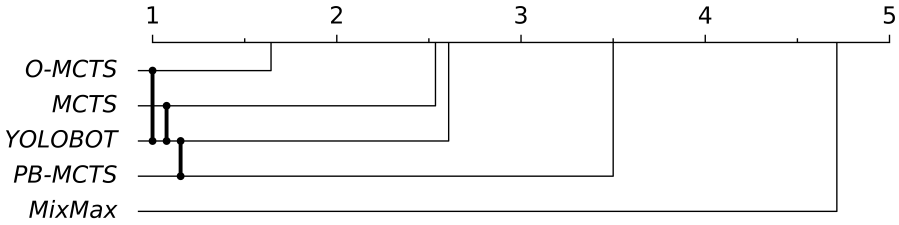
Table 7.1: The games used in the experiments

Game	Time	Borda-MCTS	MCTS	Yolo-bot	PB-MCTS	MixMax
Jaws	10^4	**100%**	**100%**	27.5%	80%	67.5%
		1083.8	832.7	274.7	895.7	866.8
	10^3	92.5%	**95%**	35%	52.5%	65%
		1028.2	958.9	391.0	788.5	736.4
	500	85%	**90%**	65%	50%	52.5%
		923.4	**1023.1**	705.7	577.6	629.0
	250	**85%**	**85%**	32.5%	37.5%	37.5%
		1000.9	997.6	359.6	548.8	469.0
Surround	10^4	**100%**	**100%**	**100%**	**100%**	**100%**
		81.5	71.0	81.2	64.3	57.6
	10^3	**100%**	**100%**	**100%**	**100%**	**100%**
		83.0	80.8	77.3	40.8	25.0
	500	**100%**	**100%**	**100%**	**100%**	**100%**
		84.6	61.8	83.3	26.3	17.3
	250	**100%**	**100%**	**100%**	**100%**	**100%**
		83.4	64.7	76.1	14.3	10.3
Aliens	10^4	**100%**	**100%**	**100%**	**100%**	**100%**
		82.4	81.6	81.5	81.8	77.0
	10^3	**100%**	**100%**	**100%**	**100%**	**100%**
		79.7	78.4	**82.2**	76.9	76.4
	500	**100%**	**100%**	**100%**	**100%**	**100%**
		78.0	77.3	**81.1**	77.2	76.0
	250	**100%**	**100%**	**100%**	**100%**	**100%**
		77.7	77.1	**79.3**	75.8	74.8
		⋮	⋮	⋮	⋮	⋮

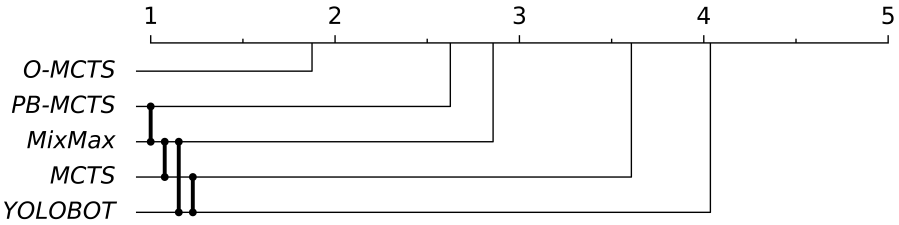
Game	Time	Borda-MCTS	MCTS	Yolo-bot	PB-MCTS	MixMax
Chase	10^4	**87.5%**	80%	50%	67.5%	37.5%
		6.2	6.0	4.8	5.2	3.9
	10^3	60%	50%	**70%**	30%	17.5%
		4.8	4.8	**5.1**	3.7	2.6
	500	55%	45%	**90%**	27.5%	12.5%
		4.9	4.5	**5.5**	2.9	2.1
	250	40%	32.5%	**90%**	17.5%	7.5%
		4.2	4.1	**5.6**	2.5	2.6
Boulderchase	10^4	62.5%	75%	45%	**82.5%**	30%
		23.7	22.1	18.8	**27.3**	20.1
	10^3	50%	32.5%	**52.5%**	40%	22.5%
		22.8	18.6	21.8	18.1	16.2
	500	**47.5%**	30%	35%	32.5%	15%
		24.7	20.2	18.3	19.4	14.4
	250	40%	40%	**60%**	17.5%	15%
		20.9	20.1	**21.7**	14.7	15.3
Whackamole	10^4	**100%**	**100%**	75%	97.5%	75%
		72.5	44.4	37.0	60.1	48.5
	10^3	**100%**	**100%**	55%	77.5%	65%
		64.0	41.8	33.9	43.9	39.0
	500	**100%**	**100%**	57.5%	70%	52.5%
		59.5	50.0	29.0	38.1	35.4
	250	97.5%	**100%**	50%	65%	52.5%
		54.8	45.9	28.5	35.1	26.6
Zelda	10^4	**97.5%**	87.5%	95%	90%	70%
		8.3	7.4	3.8	**9.6**	8.1
	10^3	80%	85%	**87.5%**	57.5%	42.5%
		8.8	7.5	5.3	8.6	**8.8**
	500	62.5%	75%	**77.5%**	50%	35%
		8.6	8.2	4.6	**8.8**	7.8
	250	55%	55%	**70%**	45%	30%
		8.4	7.8	4.4	8.0	7.2
∅ Rank		1.6	2.5	2.6	3.5	4.7

Table 7.2: The results of algorithms tuned per row with average over both tables.

the average score are shown in bold, and a ranking of the algorithms is computed. The resulting average ranks are shown in the last line. We use a Friedmann test and a posthoc Wilcoxon signed-rank test as an indication for significant differences in performance. The results of the latter (with a significance level of 99%) are shown in Figure 7.2a.



(a) All game runs. Data from Table 7.2



(b) Only won game runs

Figure 7.2: Average ranks and the result of a Wilcoxon signed-rank test with $\alpha = 0.01$. Directly connected algorithms do not differ significantly.

We can see that Borda-MCTS performed best with an average rank of 1.6 and significantly better performance than MCTS and PB-MCTS. Table 7.2 allows us to take a closer look at the domains. For games that are easy to win, such as *Surround*, *Aliens*, and *Whackamole* Borda-MCTS beats MCTS and PB-MCTS by winning with a higher score. In *Chase*, a deadly but more deterministic game, Borda-MCTS can achieve a higher win rate. In deadly and stochastic games like *Zelda*, *Boulderchase*, and *Jaws* Borda-MCTS performs comparably to the other algorithms without anyone performing significantly better than the others.

Figure 7.2b summarizes the results when only won games are considered. It can be seen, that in this case, PB-MCTS is significantly better than MCTS. This implies that if PB-MCTS manages to win, it does so with a greater score than MCTS, but it wins less often. YOLOBOT falls behind because it is designed to primarily maximize the win rate, not the score.

Inspecting the performance of MIXMAX it can easily be seen that the hereby added bias towards higher scores often results in death: Looking at only won games (see Figure 7.2b) it achieves a higher rank than MCTS, but overall its performance is significantly worse. Due to Borda-MCTS beating MCTS throughout the experiments, and looking at their optimization targets (By recalling the optimization targets of Borda-MCTS and MCTS (maximizing the Borda score plays the action that beats the other alternatives most often where MCTS maximizes average performance, see Section 6.3) we can conclude

		Exploration-Exploitation								
	O-MCTS	**0**	**0.25**	**0.5**	**0.75**	**1**	**1.25**	**1.5**	**1.75**	**2**
Rollout Length	**5**	31	17	9	7	4	**1**	11	3	13
	10	39	12	6	5	14	16	2	10	8
	25	45	19	24	35	43	27	29	51	20
	50	74	59	46	61	49	55	57	69	56
	PB-MCTS	**0**	**0.25**	**0.5**	**0.75**	**1**	**1.25**	**1.5**	**1.75**	**2**
Rollout Length	**5**	63	67	71	72	73	**62**	68	64	65
	10	80	83	89	79	66	82	86	78	77
	25	97	94	91	100	101	153	104	103	92
	50	151	105	154	152	155	108	148	106	145
	MixMax	**0**	**0.25**	**0.5**	**0.75**	**1**	**1.25**	**1.5**	**1.75**	**2**
Rollout Length	**5**	146	147	98	99	96	**95**	107	102	149
	10	162	158	161	163	159	150	157	160	156
	25	174	167	173	178	165	170	166	164	172
	50	181	169	168	171	176	179	177	175	180
	MCTS	**0**	**0.25**	**0.5**	**0.75**	**1**	**1.25**	**1.5**	**1.75**	**2**
Rollout Length	**5**	53	**18**	23	30	41	22	32	28	40
	10	60	21	36	25	26	34	38	37	33
	25	76	52	42	44	54	48	50	58	47
	50	88	70	75	81	85	84	90	87	93

Table 7.3: Results for different parameters for all algorithms except for YOLOBOT (Rank 15). In each cell, the overall rank over all games and budgets is shown.

that we found evidence that Borda-MCTS's preference for actions that *maximize win rate* works better than MCTS's tendency to *maximize average performance* for the tested domains.

Parameter Optimization In Table 7.3 the overall rank over all parameters for all algorithms is shown. It is clearly visible that a low rollout length RL improves performance and is more important to tune correctly than the exploration-exploitation trade-off C. Since YOLOBOT has no parameters, it is not shown. Except for the extreme case of no exploration ($C = 0$), Borda-MCTS with $RL = 5$ is better than any other MCTS algorithm. The best configuration is Borda-MCTS with $RL = 5$ and $C = 1.25$.

Video Demonstrations For each algorithm and game, we have recorded a video where the agent wins[1]. In those videos, it can be seen that Borda-MCTS frequently plays actions that lead to a higher score, whereas MCTS play more safely—often too cautious and averse to risking any potentially deadly effect.

1 The videos are available at `https://bit.ly/2ohbYb3`

7.4 Conclusion

In this chapter, we proposed Borda-MCTS, a modification of MCTS that handles the rewards in an ordinal way: Instead of averaging backpropagated values to obtain a value estimation, it estimates the winning probability of an action using the Borda score. By doing so, the magnitude of distances between different reward signals is disregarded, which can be useful in ordinal domains. In our experiments using the GVGAI framework, we compared Borda-MCTS to MCTS, PB-MCTS, MIXMAX, and YOLOBOT, a specialized agent for this domain. Overall, Borda-MCTS achieved higher win rates and reached higher scores than the other algorithms, confirming that this approach can even be useful in domains where numeric reward information is available.

8 Handling Many Ordinal Rewards

In the past chapters, we have discussed algorithms that can handle ordinal feedback data from the environment. In Chapter 4 we have introduced PB-MCTS, an algorithm that only requires feedback in the form of pairwise comparison, but suffers from strong computational demands. To avoid these problems, we have introduced the Ordinal Agent Framework, which has a completely different way of tackling the problem: Instead of relying on dueling based bandit algorithms, we have based Borda-MCTS and Borda-UCB on voting theory. In short, these algorithms measure how often a reward is seen for different actions. Then, a voting algorithm is applied to identify the winner.

Preference-Based Monte Carlo Tree Search

Upper Confidence Bound

In our experiments, the Borda-MCTS algorithm performed much better than PB-MCTS. Sampling efficiency has been identified as a core reason for that. But that sampling efficiency comes with a cost: Samples do need to be stored and new samples are compared with old knowledge to estimate the quality of an action. The implementations for that update method used in the past chapters introduce a time complexity linearly dependent on the number of seen ordinal scores $|\hat{O}|$ (see 6.3). With a low number of rewards, this update method may never be a bottleneck, but this can not be ensured. When looking at noise in the reward signal, especially when looking at the extreme case of never seeing a reward twice, $|\hat{O}|$ could be very high or even match the number of all samples. The past sections have shown how real-valued feedback can be turned into ordinal feedback (e.g. by discarding the distance metric). Staying with that idea, we look at noisy numerical rewards, for example, received by adding Gaussian noise on the reward value. If these rewards are transferred into ordinal reward signals, the probability of seeing an ordinal reward twice reaches 0. Hence, the number of seen ordinal rewards equals the time step: $|\hat{O}| = t$. Thus, the algorithm would not be linear dependent on time, but quadratic. This would lead to a very poor performance of the Borda implementations seen in the past chapters.

In this chapter, we want to tackle the problem of bad performance of our novel Borda algorithms in the presence of many distinct reward signals. By doing so, we follow our main research question *What problems do arise? How can they be treated?*. This chapter is based upon and in parts taken from our publication [19]. The main approach is to artificially reduce the sample number of different rewards $|\hat{O}|$. If $\hat{O}$ in fact would origin from a smaller non-noisy set O, this set O would be a good reduction to work with. But there is no function given to map elements from $\hat{O}$ into O. Since neither the true rewards are known nor could they be identified, we introduce an ordinal bucketing algorithm that is capable of bucketing the noisy rewards in

a smaller set of buckets with size $y < |\hat{O}|$. Bucketing is a way to group together elements such that the number of elements (the buckets) can be assumed smaller. Concepts like generalization or clustering are very similar and can be applied here, too. Due to the nature of the past Borda algorithms, our novel bucketing algorithm needs to function online: We do not present a fixed set of noisy rewards that need to be binned, but the bucketing needs to process one sample after another updating its bucketing after each sample.

In the following, we introduce some basic concepts, a formal problem description, and different bucketing algorithms with varying complexity. Those bucketing algorithms then are evaluated on synthetically data and are applied to Borda-MCTS and tested on a noisy GVGAI task.

8.1 Reducing the Cardinality

The idea of summarizing many ordinal values to a fixed number of bins or buckets is related to many different aspects of machine learning and statistics, which we briefly survey in the following.

We start with data bucketing or binning: the concept of merging multiple value-quantity pairs to fewer interval-quantity pairs. In statistics, creating optimal histograms is a well-explored research area. Concepts of optimality are well defined and optimal solutions are known for different error measures [14]. They can often be obtained in a *first-collect-then-bucket* fashion, where one has to create a histogram for a given distribution sample, or in a streamed way where the bucketing is incrementally updated [4, 14]. In this chapter, we take a look at the featureless and ordinal way of the latter case. This is an important point since common bucketing solutions require and exploit features or a metric over the objects to bin, which makes it easy to determine which values are close to each other so that their bins can be merged. An ordinal scale, however, does not have such a metric. One cannot tell whether two ordinals are far away or close by, only which of them has a higher value. There also are ordinal clustering methods that do not require a metric but make use of features[35].

Defining the quality of a bucketing is not trivial without having a metric. A reasonable idea is to strive for buckets of equal size, which leads to q-quantiles that split a distribution into q equally sized parts, where each quantile contains $1/q$ of the complete distribution. As an example, in Borda-MCTS the root node might want to have its action rewards organized in many quantiles to get a precise view on the different action performances. A node further down the tree might not need a fine-grained split and could be fine with only storing the current median, the 2-quantile.

We propose a simple algorithm to achieve that, where the focus is on very little overhead and a short run-time since the main reason for the binning is to reduce overhead in the first place. In comparison to other bucketing or quantile approximating algorithms, our approach only

stores very little information and often needs to resort to uninformed random decisions, which nevertheless leads to good results. We test our bucketing algorithm in two distinct ways. First, we test the quantile approximation as a stand-alone algorithm for streamed data and analyze the error on the quantiles, as well as its run time and space complexity for different kinds of distribution functions. Second, we use the GVGAI Framework, as used in the evaluation in past chapters, to analyze the influence of OMCTS using this approach. As a baseline, we also compare to vanilla MCTS.

8.2 Ordinal Bucketing

In the following section, we describe three bucketing algorithms with different characteristics and how to derive quantile approximations. We first introduce the formal problem and the used bucketing structure.

8.2.1 Problem Definition

Given an unknown distribution of objects in an ordinal domain Q represented by a random variable X and a time step t, the task is to create a set H_t^X of buckets that bracket all past samples $\hat{X}_t = (X_0, X_1, ...X_t)$ together. The number of buckets $|H_t^X| \leq f(t)$ has an upper limit defined by a bound function $f(t) \in \mathbb{N}$ which is naturally smaller than t to require a real bucketing. At any given time t, the task is to create a bucketing H_t^X given the previous bucketing H_{t-1}^X and the observation X_t. The algorithm proceeds in an on-line manner, i.e., it is not possible to access all past samples $\hat{X}_h$, but only the previous bucketing which has to be updated.

A *bucket* $g = (g_u, g_n, g_d)$ is defined by an upper bound $g_u \in Q$, a number $g_n \in \mathbb{N}$ indicating the bucket size and auxiliary data g_d that can be used to calculate a pivot point of this bucket using a globally defined function $P(g_d) \in Q$. The semantic is that approximately g_n elements of $\hat{X}_t$ lie between g_u and the upper bound of the bucket below g'_u, where approximately $g_n/2$ of the buckets in g are above and below the pivot $P(g_d)$ respectively. Recall that one can not simply interpolate between the min and max values since we are on an ordinal scale.

The main idea of our dynamically adapting method is that once a bucket reaches an upper limit on g_n, it is split into two buckets, using $P(g_d)$ as their border, or, if the number of possible buckets increases, the largest bucket is split in half. As an example of bucketing, if one wants to calculate the 2-quantile (the median) of a data stream, you use two buckets. The upper bound of the lower bucket represents the median. Asking for a 3-quantile, one needs three buckets, and so on.

8.2.2 Bucketing Algorithms

In the following, we explain three novel ordinal bucketing methods. Every value $o \in Q$ is assigned to exactly one bucket g_o. For convenience, we introduce the following notations: $N(o)$ is the number of stored values of the bucket g_o, $U(o)$ is its upper bound, and $D(o)$ is the data stored in g_o. Later, the number of stored elements $N(o)$ of g_o is updated by calling *Store o*, indicating that the reward o has been seen once more.

First-n-Bucketing

A simple algorithm for bucketing ordinal values takes the first $n-1$ distinct ordinal values and uses them as upper bounds for its buckets. Bucket n stores all further rewards that exceed the other upper bounds. Hence, the upper bound function is independent of the number of seen samples: $f(t) = n$. We call this approach First-n-Bucketing (see Algorithm 5). It is not capable of dynamically increasing the number of buckets and will be used as a baseline later. This algorithm does not store any auxiliary data g_d nor does it update the bucket bounds. This is a very simple idea with

Algorithm 5 Adding a value with First-n-Bucketing

Require: Time t, Sample X_t, Previous Bucketing H^X_{t-1}, Number of Buckets n
  if $|H^X_{t-1}| < n$ **and** $U(X_t) \neq X_t$ **then**
    $H^X_t = H^X_{t-1} \cup \{(X_t, 0, \emptyset)\}$
  end if
  Store X_t

k-Log-Growing

The next idea addresses the dynamically increasing number of buckets. Here, the number of buckets has a logarithmic bound on the number of seen samples: $f(t) = k \log(t)$ with k being a parameter to scale the number of buckets. We named the resulting algorithm k-Log-Growing (see Algorithm 6).

In the initialization phase, an empty bucket spanning the complete range is added. In the beginning, new samples are added to this one bucket. Once the upper bound of available buckets $f(t) = k \log(t)$ increases, a new bucket can be added. Instead of adding a new empty bucket, we split an existing bucket using a pivot point.

Why don't we add new buckets?

Simply adding a new empty bucket would result in problems: The new bucket does not contain any values, despite the fact, that there could very well be already stored values in its range, and the new

upper bound does not utilize any information about the general distribution of the data, we might have gathered so far.

For computing the pivot point of a bucket $g = (g_u, g_n, g_d)$, this algorithm uses the auxiliary data $g_d \in Q^m$ to store the last m values seen in this bucket, where m is odd. These m data points can be used to compute an approximate median. Empirically, we have found that $m = 3$ is enough to show decent behavior. If a bucket has not yet seen m data points, the pivot can not be computed. We refer to this with '*has pivot*' in the following algorithms. The auxiliary data of a bucket is updated, whenever a new sample is added into the bucket with *Store* X_i and replaces the oldest entry.

An obvious choice for the bucket to be split is the largest bucket since we try to get equally sized buckets. Its pivot point is used as the splitting point, which results in two equally sized buckets, the lower one having the previous pivot point as its upper bound, and the other re-using the previous upper bound. If the largest bucket has too few seen samples to estimate the pivot, it is not split directly but one waits until it has enough data to do so.

Since after initialization all elements are added to the first bucket and splitting is only affected by the last m seen samples, it is easily possible to create non-optimal splits. Especially for streams with a low number of different values (which are repeated often), this initialization could result in an arbitrarily bad bucketing.

k-log-Growing-First-n

Combining the two previous ideas, we get an algorithm that applies bucketing only after n distinct ordinal values have been observed and then increases the number of buckets, dependent on the number of observed values. This algorithm, k-Log-Growing-First-n (see Algorithm 7), boosts the accuracy for a few observed values, while it is still able to handle large amounts of data.

If we take a look at the space-complexity of our presented algorithms (see Fig. 8.1), the decrease from $\mathcal{O}(t)$ for no bucketing to $\mathcal{O}(\log t)$ for k-Log-Growing-Bucketing, respectively $\mathcal{O}(1)$ for First-n-Bucketing, is huge. For time-complexity, we assume a data-structure with logarithmic reading and writing complexity, resulting in $\mathcal{O}(t \log t)$ for adding

Algorithm 6 Adding an ordinal value with k-Log-Growing

Require: Time t, Sample X_t, Previous Bucketing H^X_{t-1}, Parameter k
  if $t == 0$ **then**
    Initialize with an empty bucket
  end if
  Store X_t
  if $|H^X_{t-1}| < k \log(t)$ **and** the largest bucket has a pivot **then**
    Split the largest bucket
  end if

Algorithm 7 Adding an ordinal value with k-Log-Growing-First-n

Require: Time t, Sample X_t, Previous Bucketing H^X_{t-1}, Parameter k, Parameter k
  if $|H^X_{t-1}| < n$ **and** $U(X_t) \neq X_t$ **then**
    $H^X_t = H^X_{t-1} \cup \{(X_t, 0, \emptyset)\}$
  end if
  Store X_t
  if $|H^X_{t-1}| \geq n$ **and** $|H^X_{t-1}| > k \log(t)$ **and** largest bucket has pivot **then**
    Split the largest bucket
  end if

values with no bucketing versus $\mathcal{O}(t \log \log t)$ and $\mathcal{O}(t)$ for adding values with k-Log-Growing-Bucketing, respectively First-n-Bucketing. Splitting a single Bucket in k-Log-Growing-Bucketing has a complexity of $\mathcal{O}(\log t)$, since it has to iterate over every bucket to find the smallest one, and is performed $\log t$ times, resulting in $\mathcal{O}((\log t)^2)$.

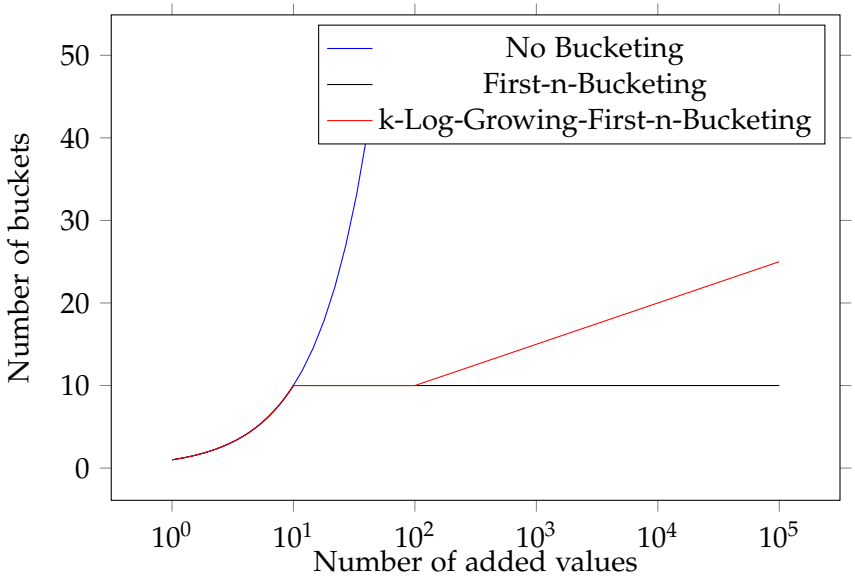


Figure 8.1: Space-complexity for no bucketing, First-n-Bucketing and k-Log-Growing-First-n-Bucketing

8.3 Analysis of Bucketing Error

In the past section, we have introduced three novel bucketing algorithms. In this section, we start the evaluation of those using synthetical data streams. In the next section will install the bucketing algorithms in O-MCTS and perform evaluations using the GVGAI-Framework.

8.3.1 Experimental Setup

We first analyze the performance of our on-line bucketing algorithm. To do so, we assume a single stream of ordinal rewards directly sampled from a true distribution. Our bucketing can not answer queries for a sample probability of a given value o, since for each bucket only the stored number of samples, an upper and a lower bound is known. Hence, it is impossible to measure common error terms like the *Sum of Squared Errors*, a common error measure. We instead compare our bucketing to the q-quantiles, where q is the number of buckets. To measure the difference, we look at the n-th bucket's upper bound and compare it to the n-th q-quartile. Let u_n be the upper bound of bucket n, q_n the n-th q-quartile, $Rank(o)$ the rank of the value o (the number of samples with a lower value than o), and t the complete sample size. We measure the distance of a bucketing to the q-quartiles using:

$$E(H) = \frac{1}{q} \sum_{1}^{q} \frac{|Rank(u_n) - Rank(q_n)|}{t}$$

For the following part of the experiment, we use a Gaussian distribution (unless mentioned otherwise) for sampling and average the results over 100 runs. First, we measure E depending on m for different values of k in k-Log-Growing-Bucketing with 1000 sequentially added samples (see Fig. 8.3). Since $m = 3$ appeared to be a reasonable choice, we use it in the following experiments. Next, we compare the error for different values of n and k in k-Log-Growing-First-n-Bucketing to see the influence of using the first n observed values as buckets, also averaged over 1000 runs (see Fig. 8.4). Each distribution is tested for an increasing number of samples to detect potential distribution-dependent behavior of k-Log-Growing-First-n-Bucketing ($n = 5, k = 2$) (see Fig. 8.7).

8.3.2 Results

In these experiments, a single bucketing is used to bucket a bigger stream of data (up to 10^5). Figure 8.3 shows a decrease of the error E from $m = 1$ to $m = 3$, while the behavior for $m > 3$ does increase again for all settings except for $k = 1$. Hence, storing the last three values is a decent choice for the algorithm compared to the other tested alternatives. As explained in the previous section, the following tests are performed using $m = 3$.

Figure 8.4 shows that for $k > 1$ the error E is fairly independent of n. The sharp increase for $k = 1$ can be explained by the fact, that after 7 or more initial buckets, no further splits are performed for 1000 added samples, resulting in the same behavior as First-n-Bucketing. Overall, it seems to be the case that for $n \to k \log T$, where T is the total number of added values, the error increases, which also explains the slight upward trend for $k = 2$ and $n \geq 9$. If any assumptions

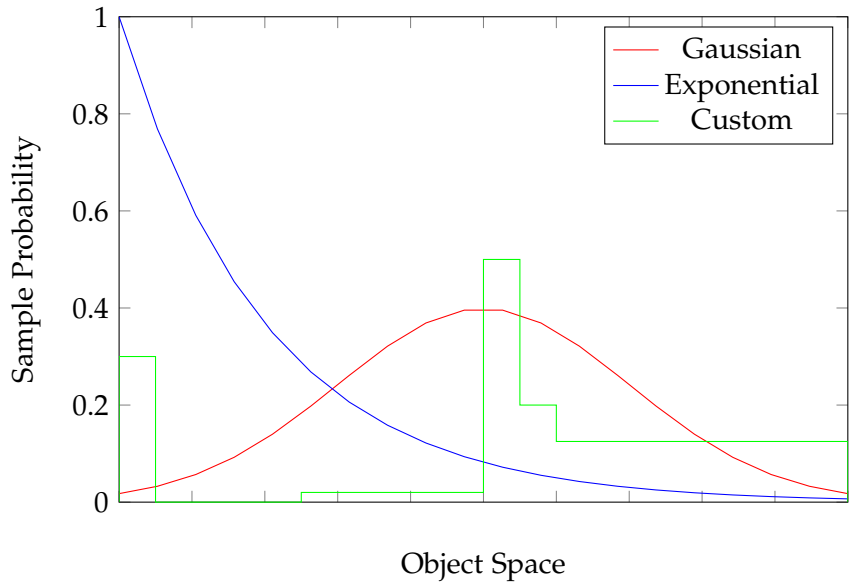


Figure 8.2: The three distributions, used in the experiments.

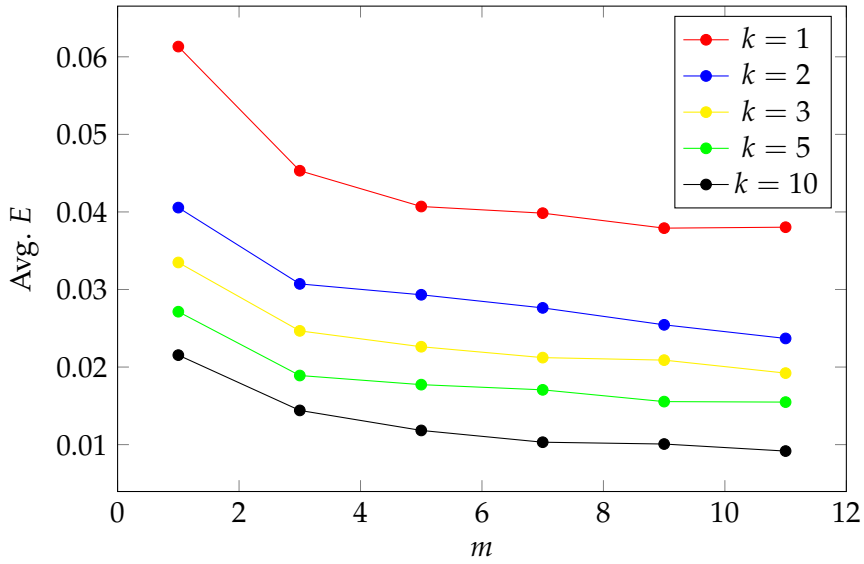


Figure 8.3: Average distance to the true percentiles for k-Log-Growing-Bucketing with different values of k, dependent on m. Results for 1000 added values, averaged over 100 runs.

regarding T are possible, this information could be used to tune n respectively. But in our case, we decided to go for a value of $n = 5$ to separate k-Log-Growing-First-n-Bucketing from the simple k-Log-Growing-Bucketing, while avoiding a strong, n-induced increase of error.

The experiments confirm the intuition, that a larger number of buckets, induced by k, results in a smaller error E (cf. Figures 8.3, 8.4 and 8.5). Since this behavior was expected, it also justifies the error-measure itself. Figure 8.5 also shows a fairly stable trend, once the

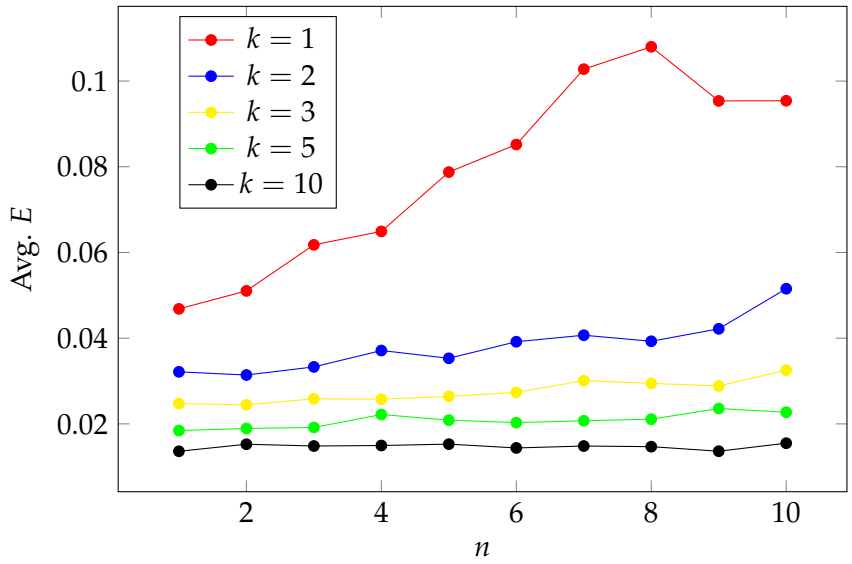


Figure 8.4: Average distance to the true percentiles for k-Log-Growing-First-n-Bucketing with different values of k, dependent on n. Results for 1000 added values, averaged over 100 runs.

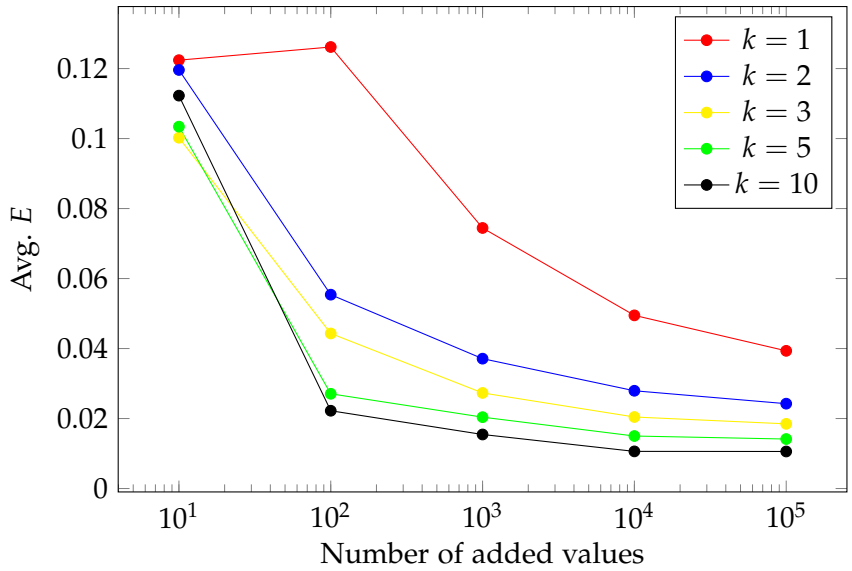


Figure 8.5: Average distance to the true percentiles for k-Log-Growing-First-n-Bucketing with different values of k, dependent on the number of added values. Results for $n = 5$, averaged over 100 runs.

first 100 values have been added. The lack of improvement for $k = 1$ between 10 and 100 derives from the same problem ($n \to k \log T$), we described earlier.

Figure 8.6 shows a steady error for the First-n-Bucketing, while the k-Log-Growing-Bucketing and k-Log-Growing-First-n-Bucketing converge to a similar, lower error. The idea of using the first n values as a foundation for further splits doesn't seem to help, since it induces the initial high error of the First-n-Bucketing. This doesn't affect the per-

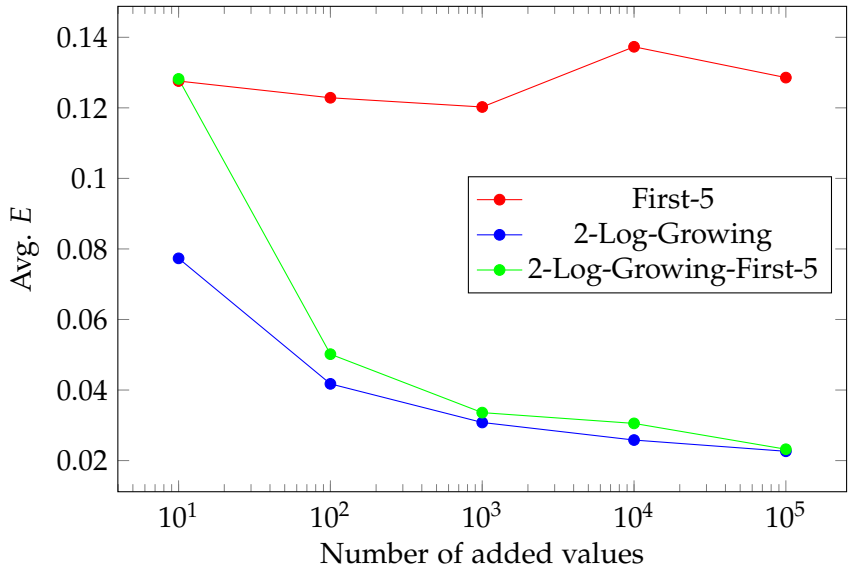


Figure 8.6: Average distance to the true percentiles for instances of the three presented algorithms, dependent on the number of added values. Results averaged over 100 runs.

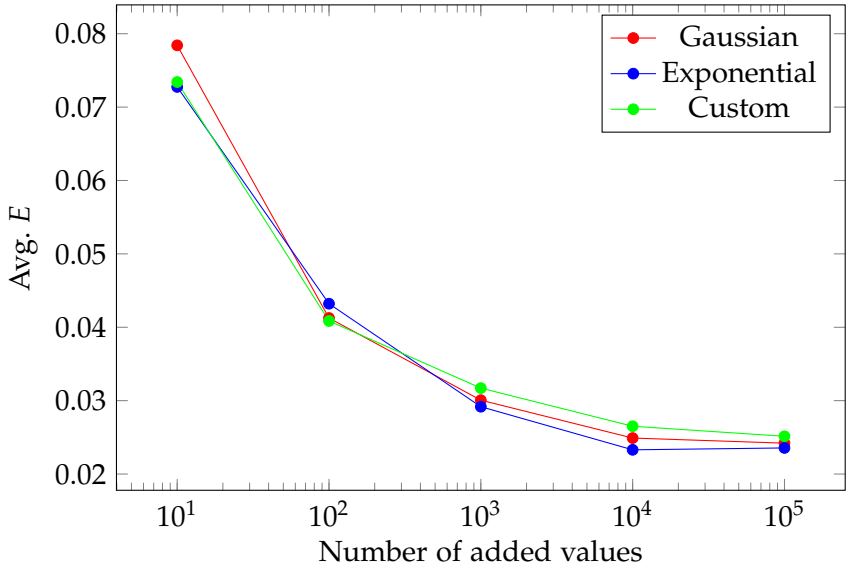


Figure 8.7: Average distance to the true percentiles for three different distribution, dependent on the number of added values. Results for k-Log-Growing-First-n-Bucketing with $k = 2$ and $n = 5$, averaged over 100 runs.

formance for large amounts of values, but k-Log-Growing-Bucketing generates a strictly lower error, which also matches the results in Figure 8.4, where no error-decrease could be seen with an increasing n.

Finally, Figure 8.7 exhibits an almost identical performance of k-Log-Growing-First-n-Bucketing on all three used distributions, indicating good robustness.

8.4 Bucketing With OMCTS

After defining how we dynamically approximate quantiles for a stream of data, we now show how to integrate bucketing into OMCTS. To this end, we take a look at how different actions are rated locally in a given node.

In vanilla OMCTS, the complete estimated probability distribution functions $f_a(o)$ for each action a and value o are stored and updated. Given two actions a_i and a_j one can estimate the probability

$$\Pr(a_i \succ a_j) = \Pr(a_i > a_i) + \frac{1}{2}\Pr(a_i = a_i) \tag{8.1}$$

using the Borda score as introduced in Section 6.2. The linear complexity of OMCTS arises from estimating pairwise comparisons over all arms and ordinal rewards. Using our bucketing method, we can reduce this complexity. Instead of storing and iterating over the complete list of seen ordinal values, we now only store and iterate over the stored buckets. For an action a and its bucketing H^{X_a} with s_a buckets and s_t aggregated values, a bucketed distribution function $\hat{f}_a(o)$ can be derived. In our experiments, we have used the upper bound g_u as a representative value for a given bucket g and the relative proportion of this bucket $g_r = g_n/s_t$ as its sample probability. Therefore, we interpret the buckets as if only the representative value would have been seen with the cumulative sample probability of all values in this bucket. Hence, for a bucket g it holds that $\hat{f}_a(g_u) = g_r$ and $\hat{f}_a(o) = 0$ for all other values o. Finally, we have used $\hat{f}$ instead of f to estimate $\Pr(a_i \succ a_j)$ for the bucketed OMCTS versions in the following experiments.

8.4.1 Experimental Setup

First, we analyze the performance of bucketed OMCTS on GVGAI games in comparison to OMCTS without bucketing and plain MCTS. We compare the quality of play of these algorithms along the following dimensions: win rate, achieved score, and the average number of iterations. The win rate is the most important measure since the very first task of a game-playing agent is to win games. The second-order task is to win with a high score. We also inspect the average number of iterations per turn to analyze the run time complexity of different approaches. To tackle non-determinism, we average those values over 100 experiments. As default for GVGAI, an agent has 40 milliseconds to choose an action. Additionally, we also test agents with 200 ms to see the results for higher sample sizes. Additionally, we repeat these experiments by disturbing the obtained rewards with artificial Gaussian noise with standard deviations of 0.1, 1, and 10, which essentially has the effect that no reward is seen more than once, and bucketing becomes crucial for a good performance.

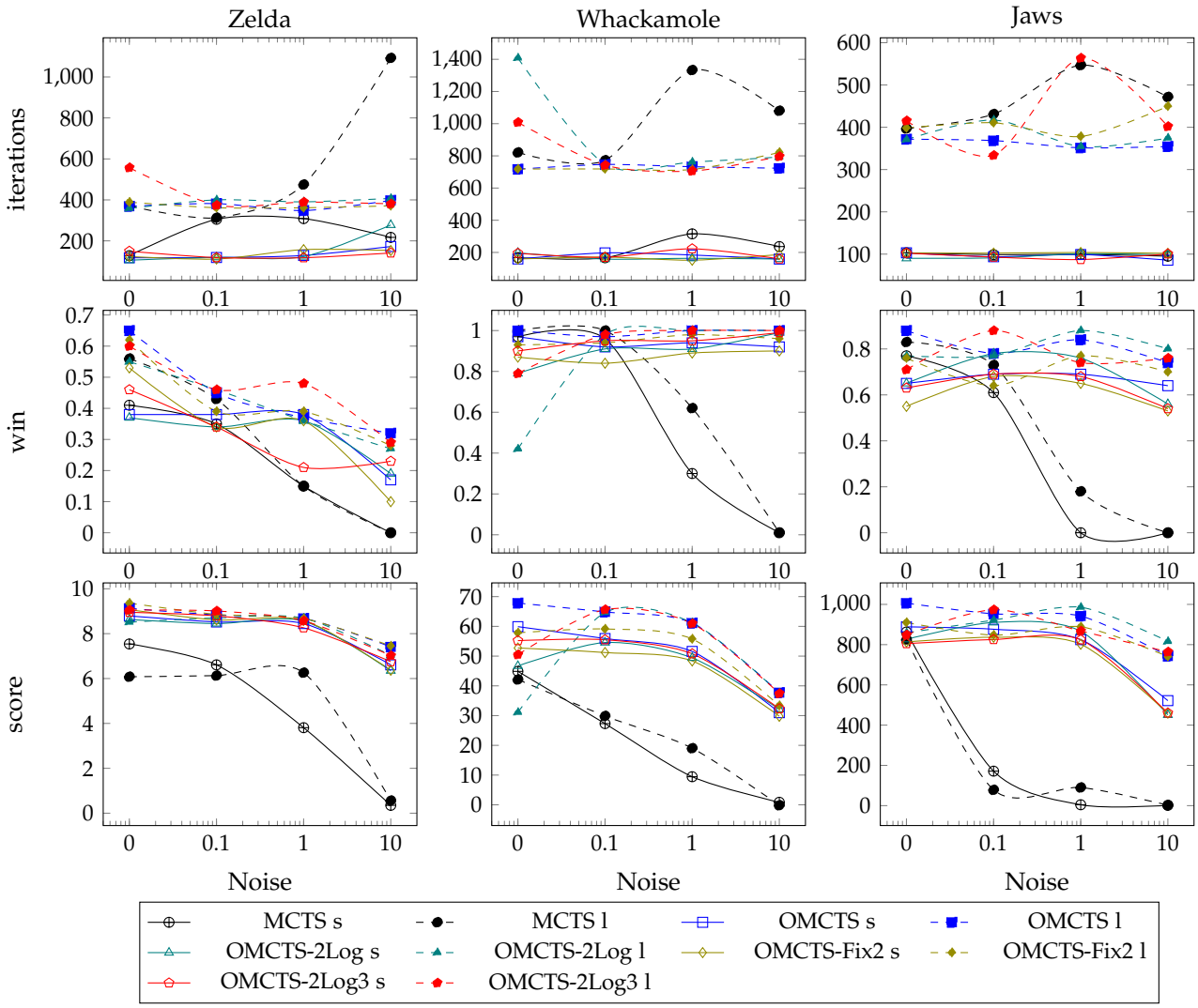


Figure 8.8: The results, scores and iterations of all five algorithms on three games having 20ms (*s* postfix) and 400ms (*l* postfix)

The tested algorithms are MCTS, OMCTS, and OMCTS with the three different bucketing methods introduced in Section 8.2: OMCTS-Fix2 (using First-2-Bucketing), OMCTS-2Log (using 2-Log-Growing-Bucketing), and OMCTS-2Log3 (using 2-Log-First-3-Growing-Bucketing) as described in the last section. These five algorithms are tested on three GVGAI games, *Zelda*, *Whackamole*, and *Jaws*. Each of these games has interesting characteristics to test on. For a description of the games, see Table 7.1.

8.4.2 Results

Figure 8.8 shows the average number of iterations per turn and the average win and score values per game dependent on different noise levels. Over all games, one can see that the number of wins, the

score, and the number of iterations does not differ significantly for different bucketing versions. Even using no bucketing does not show a significant difference.

Monte Carlo Tree Search

The most outstanding result is how the real-valued MCTS and B-MCTS versions behave in the presence of noise: The performance of MCTS constantly drops when the noise is increased having nearly zero wins at an SD of 10. Even though this is not unexpected at such a high noise level, OMCTS is, on the other hand, still able to perform well. Thus, it seems to be much more robust against noisy rewards because although the obtained score decreases, the winning chances of OMCTS do not suffer as much as those of MCTS. In *Zelda*, OMCTS seems to struggle to find the key, which, it being a prerequisite for winning the game, results in fewer wins. For *Jaws*, the number of wins stays unsteady but constantly in the area of 60% to 90%. In *Whackamole*, one even can see an increase in wins given noise. A reason for that is that colliding with the cat can still be evaded (since losing is a high negative reward) and additionally, the agent is not lured into dangerous positions (where a mushroom is right next to the cat) since the noise conceals these positive rewards. The only outstanding point looking at OMCTS variants is the bad performance of OMCTS-2Log in *Whackamole*. As mentioned in Section 8.2, there is a chance of this algorithm to fail due to a bad initialization. One can see that this does not happen in the presence of noise since the chances of seeing the same reward twice go to zero.

Looking at the average number of iterations, one can see a direct correlation between iterations and lost games. Sadly the number of iterations that the agent can perform seems to differ heavily with the current state of the game and whether terminal states are often sampled or not. Hence, we can not deduce a significant difference of iterations whether bucketing is used or not. Most of the time the number of iterations and hence also rewards are below 4000, even for the extended 200 ms turns. This number seems to be too small to make a significant time-saving in contrast to the expense of the GVGAI framework itself. Furthermore, it is even more interesting to see, that for iteration numbers below 500 the use of bucketing does not decrease the performance. This could be easily possible since bucketing in each node induces an overhead that only is significant for very low stored samples.

8.5 Conclusion

We have proposed an ordinal bucketing method that separates a stream of data into multiple buckets, where the number of buckets increases with the amount of available data. Since ordinal values do not allow the use of distance-based error measures, the bucketing strategy tries to keep the buckets filled equally, leading to quantile estimation.

Our results show that the proposed k-log-first-n-bucketing has a good runtime and quality of play for both small and large amounts of data. Using the GVGAI framework we show interesting results comparing OMCTS with and without bucketing: While both have an overall good performance, OMCTS shows a completely different behavior than MCTS in the presence of noise, where MCTS fails to win games and OMCTS loses score but mainly can keep the number of won games.

9 Ordinal Reinforcement Learning

In the past chapters, we have investigated how well known multi-armed bandit and Markov Decision Process algorithms can be modified to support ordinal reward values. Following the main research question for this thesis, we investigate how those results can be translated into related problem descriptions. In this chapter, we apply the successfully tested ordinal framework and Borda technique in the domain of *Reinforcement Learning* (RL). Comparing MCTS to RL, the main difference is the absence of a forward model for RL. Hence, it is not possible to do rollouts or simulated moves, but each move needs to be executed directly in the environment. The common RL task is to learn a good policy using multiple runs or executions - which are called episodes in this domain. Instead, the common MCTS task would be to perform as well as possible in the very first episode, without the explicit need to learn or aggregate knowledge over multiple episodes or action executions. This chapter is based on our publication [53].

Monte Carlo Tree Search

9.1 Related Work

Recently, there have been several proposals for combining preference learning with RL, where pairwise preferences over trajectories, states, or actions are defined and applied as feedback signals in reinforcement learning algorithms instead of the commonly used numerical rewards. For a survey of such preference-based reinforcement learning algorithms, we refer the reader to [46].

While preference-based RL provides algorithms for learning an agent's behavior from pairwise comparison of trajectories, [45] presents an approach for creating preferences over multiple trajectories in the order of ascending ordinal reward tiers, thereby deviating from the concept of pairwise comparisons over trajectories. Using a tutor as an oracle, this approach approximates a latent numerical reward score from a sequence of received ordinal feedback signals. This alternative reward computation functions as a reward transformation from the ordinal to the numerical scale and are applicable on top of an existing reinforcement learning algorithm.

Contrary to this approach, we do not use a tutor for the comparison of trajectories but can directly use ordinal rewards as a feedback signal. To use environments where numerical feedback already exists without the need for acquiring human feedback about the underlying preferences, we automatically extract rewards on an ordinal scale from existing environments with numerical rewards. To this end, we adapt

the Borda approach presented in Chapter 6 and implemented to MCTS in Chapter 7 to RL.

Furthermore, we handle ordinal rewards in a similar manner as previous approaches by directly using aggregated received ordinal rewards for comparing different options. The idea of direct comparison of ordinal rewards builds on the works of [43], [44], [12] and the previous chapters, which provide criteria for the direct comparison of ordinal reward aggregations.

In summary, we automatically transfer numerical feedback into preference-based feedback and propose a new conceptual idea to utilize ordinal rewards for reinforcement learning, which should not be seen as an alternative for the existing algorithms stated above. Hence, we do not compare the performance of our new approach to any of the algorithms that use additional human feedback, but to common RL techniques that use numerical feedback.

9.2 Ordinal Reinforcement Learning Approach

Markov Decision Process
Ordinal Agent Framework

In this section, MDPs and RL algorithms are adapted to settings with ordinal reward signals. More concretely, we use the OAF and apply the same technique as used for Borda-UCB1 and Borda-MCTS and explain how this method can be used in Q-Learning and Deep Q-Networks in order to learn to solve environments that return feedback signals on an ordinal scale. By doing so, we describe the implementation of the OAF by presenting the pseudocode for the algorithm and showing the OAF method implementations, just like it has been done in the previous chapters.

9.2.1 Aggregation Approach

Before defining how the OAF is used in detail, we first show the approach of our ordinal RL algorithm and its aggregation step in detail. In order to aggregate multiple ordinal rewards, an empirical cumulative density function $\hat{F}$ is used to store and represent the expected frequency of received rewards on the ordinal scale (compare to Section 6.3). This distribution is represented by $\hat{F}_{(s,a)}(o)$, which is the empirical frequency of receiving an ordinal reward less or equal to o by executing action a in state s. As also done in previous chapters, we use $\hat{f}_{(s,a)}(o)$ to represent the empirical probability of receiving the exact ordinal reward o by executing action a in state s.

Internally, a counting vector is used to store how often a reward o_i has been seen at a state transition using action a in state s:

$$G(s,a) = \begin{bmatrix} g_{o_i}(s,a) \\ ... \\ g_{o_u}(s,a) \end{bmatrix} \tag{9.1}$$

Through normalization of the counting vector G, the empirical density function $\hat{f}$ (and $\hat{F}$) can be constructed, which represents the expected probability of receiving a reward equal to (or $\leq$ than) the given ordinal reward o_i:

$$\hat{f}_{(s,a)}(o_i) := \frac{g_{o_i}(s,a)}{\sum_{j=0}^{u} g_{o_j}(s,a)} \tag{9.2}$$

$$\hat{F}_{(s,a)}(o_i) := \sum_{j=0}^{i} \hat{f}_{(s,a)}(o_j) \tag{9.3}$$

This shows how one can compute $\hat{F}$ or $\hat{f}$ based on G. In the following, we show how G can be updated. Note that this will result in changes to $\hat{F}$ or $\hat{f}$ since they are defined on G.

Value Function For Ordinal Rewards

While numerical rewards enable the representation of value function $V_\pi(s)$ by the expected sum of rewards, the value function for environments with ordinal rewards needs to be estimated differently. Since ordinal rewards are aggregated in a distribution of received ordinal rewards $\hat{F}$, the calculation of value function $V_{\pi,\hat{F}}(s)$ in state s can be done based on $\hat{F}$. Hence, the computation of the value function can be modeled by the following formula of

$$V_{\pi,\hat{F}}(s) = \bar{B}_{\hat{F},A(s)}(s,a) \text{ with } a = \pi(s). \tag{9.4}$$

The computation of the value function from probability distribution $\hat{F}$ through function $\bar{B}$, can be done using the Borda-Score (see Section 6.2 and compare Section 6.3). Here, just like for the ordinal variants of MCTS and UCB, one could use other alternatives like the Condorcet winner to define the winner of voting. For the scope of this thesis, we continue using the Borda Score.

The resulting Borda-Score of an action a computes the probability that action a receives a better ordinal reward than a random alternative action a' in the same environmental state s. Following the definition of Borda Score from Section 6.2, this probability is $\hat{B}_{F,A(s)}(a)$. Hence, the value function of policy π at state s can be calculated based on $\hat{F}$ using the Borda Score: $V_{\pi_{\hat{F}},\hat{F}}(s) = \hat{B}_{\hat{F},A(s)}(a)$ with $a = \pi_{\hat{F}}(s)$. Further, a policy maximizing the Borda Score using $\hat{F}$ is available with $\pi_{\hat{F}}(s) = \arg\max_{a \in A(s)} \hat{B}_{\hat{F},A(s)}(a)$.

Based on Formula 9.4, the optimal policy $\pi^* = \pi_F$ (with F being the true, non-empirical density function) can be determined in the same way as for numerical rewards by maximizing the respective value function $V_{\pi_F,F}(s)$.

9.2.2 Transformation of existing numerical rewards to ordinal rewards

If an environment has pre-defined rewards on a numerical scale, transforming numerical rewards $r \in \{r_1, ..., r_u\} \subset \mathbb{R}$ into ordinal rewards

$r_o \in \{o_1, ..., o_u\}$ can easily be done by translating every numerical reward to its ordinal position within all possible numerical rewards. This way the lowest possible numerical reward is mapped to o_1, and the highest numerical reward is mapped to position o_u, with u representing the number of possible numerical rewards. This transformation process simply results in removing the metric and semantic of distances of rewards but keeping the order. A similar transformation has been done for Borda-MCTS in Section 7.3.1 but for two-dimensional rewards.

9.3 Ordinal Reinforcement Learning

In Section 9.2.1, we have shown a possibility on how to compute a value function $V_{\pi_{\hat{F}}}(s)$ and defined the optimal policy π^* for environments with ordinal rewards. In this section, this is used to adapt common reinforcement learning algorithms to ordinal rewards. In the following, we introduce new ordinal algorithm variants for Q-Learning and DQN. First, we introduce our novel algorithm variants followed by the implementation of both algorithms in our OAF to highlight the algorithm differences[1]. We do that for both, Q-Learning and DQN in that order. To show that the ordinal modifications are not only applicable for baseline algorithms, we also apply common modifications to our ordinal DQN algorithms like experience replay, target and evaluation networks, and double deep q networks.

9.3.1 Ordinal Q-Learning

For the adaptation of the Q-Learning algorithm to ordinal rewards, we do not directly update a Q-value $Q(s, a)$ that represents the quality of a state-action pair (s, a) but update the empirical density function $\hat{F}$ of received ordinal rewards. The target distribution is computed by adding the received ordinal reward o_i (represented through unit vector e_i of length u) to the distribution $G(s', \pi^*(s'))$ of taking an action in the new state s' according to the optimal policy π^*. The previous distribution $G(s, a)$ is updated with the target distribution by interpolating both values with learning rate α, which can be seen in the formula

$$G(s, a) = G(s, a) + \alpha[e_i sa + \gamma G(s', \pi^*(s')) - G(s, a)] \tag{9.5}$$

In this adaptation of Q-Learning[2], the expected quality of state-action pair (s, a) is not represented by the Q-value $Q(s, a)$ (see Formula 2.6) but by the function $\bar{B}$ (see Formula 6.3) of the probability distribution $\hat{F}$, which is derived from the iteratively updated distribution $G(s, a)$.

1 For descriptions of vanilla algorithms, we refer to Section 2.6.1 and Section 2.6.2

2 This technique of modifying the Q-Learning algorithm to deal with rewards on an ordinal scale can analogously be applied to other Q-table based reinforcement learning algorithms like *Sarsa* and *Sarsa-λ* [52]

Algorithm 8 The Q-Learning Algorithm using the OAF

Input : The start state s_0
Output : A learned policy π.

$D \leftarrow$ *empty aggregation data*
$s \leftarrow s_0$
while computation time left **do**
  $a_i, s', r \leftarrow \text{OAF}_{MCTS}.Sample(D, s)$
  $\text{OAF}_{MCTS}.Update(D, s, a_i, r, s')$
  $s \leftarrow s'$
  if s is terminal **then**
    $s \leftarrow s_0$
  end if
end while
return $\pi(s) = \arg\max_a GetRatings(D, s, a)$

The baseline implementation of vanilla Q-Learning is presented in the following. There, one can see how a basic RL algorithm can be implemented using our novel Framework. It also forms the base of the Ordinal Q-Learning implementation that only needs to modify certain parts of the algorithms (See the next paragraph).

As shown in Section 5 and used in previous framework implementations, we define the algorithm by giving pseudocode for the overall control flow and providing implementations for the functions needed in OAF. The Q-Learning algorithm can be written as an instance of the OAF with the following implementations used by Algorithm 8:

1. The Aggregator state at time t is the current Q-Table $D_t = Q_t(s, a)$.
2. The Aggregator Update: $D_t, s_t, a_t, r_t, s_{t+1}$ is implemented using the Bellman Equation:
   $Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[r_t + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)]$
   with other values being carried over: $Q_{t+1}(s, a) = Q_t(s, a)$.
3. Get Ratings only needs to point to the Q-Value of an action: $V_t(s, a) = Q_t(s, a)$
4. Agent.ChooseAction picks the action maximizing the Q-Value: $a_t = \arg\max_a V_t(s(a), a)$ with $s(a)$ being the unique state of a.

Having that implementation, we show that this algorithm equals the definition of Q-Learning algorithm from Section 2.6.1. First, we notice that Q-Learning - as well as all following RL algorithms but different to MCTS and UCB - iterate over multiple episodes. The only data stored in Q-Learning is the Q-Table and as defined above, this is D_t at time t. Hence, we can realize multiple episodes by restarting the algorithm and carry over the Aggregator data D from the last episode. This way the Q-Table stays persistent over the course of episodes.

The next thing to check is how new information is stored in the Q-Table Q. As set in 2.5, the bellman equation is used to update the Q-Table and the formula is literally the same as for Q-Learning: The state-transition feedback is used together with the future state information to update the current actions Q-Value, leaving other values untouched. The used policy for Q-Learning is to play the action with the highest Q-Value, which is exactly done here too, since Agent.ChooseAction is called with all actions available in the current state (see 2).

Having covered all that, we argue that the framework implementation stated above equals Q-Learning.

The ordinal Q-Learning implementation can reuse the above implementation and modify the aggregation step to utilize the Borda-Score as described at the beginning of Section 9.3.1. There are only a few parts that need to be changed in comparison to the baseline Q-Learning Framework implementation: The aggregation step does not resemble the Q-Table, but the reward distribution G per state-action pair. Going further, the rating function needs to be adapted, too, since there are no Q-values that can directly be used. Instead, we use the Borda-Score that is calculated using the sampled ordinal rewards G as introduced in Section 9.2.1. This leads to the following algorithmically changes:

1. The Aggregator state at time t is the current sampled reward distribution $D_t = G$.
2. The Aggregator Update: $D_t, s_t, a_t, r_t, s_{t+1}$ updates G with the newly seen reward and the distribution of the next state, as defined in Formula 9.5. The other values stay untouched: $D_{t+1}(s, a) = D_t(s, a)$
3. The Get Ratings function now uses G to calculate the Borda-Score of the given actions based on their rewards stored in D_s: $V_t(s, a) = \hat{B}_{A(s),G}(a)$
4. Agent.ChooseAction picks the action maximizing the Q-Value: $a_t = \arg\max_a V_t(s(a), a)$ with $s(a)$ being the unique state of a.

9.3.2 Deep Q-Network

As introduced in Section 2.6.2, the idea of *Deep Q-Learning* (DQL) is to learn a function that predicts Q-Values instead of having a lookup table as used by vanilla Q-Learning. The Deep Q-Network Algorithm uses a DNN to model this function. The overall algorithm used by DQL is the same as for Q-Learning, since only the way how Q-Values are stored and updated is changed. Hence, the following shows how the DQL algorithm is modeled using the OAF together with Algorithm 8[3]:

3 Further details, like the parameters or setup of the DNN are neglected here and will be stated in the experiments

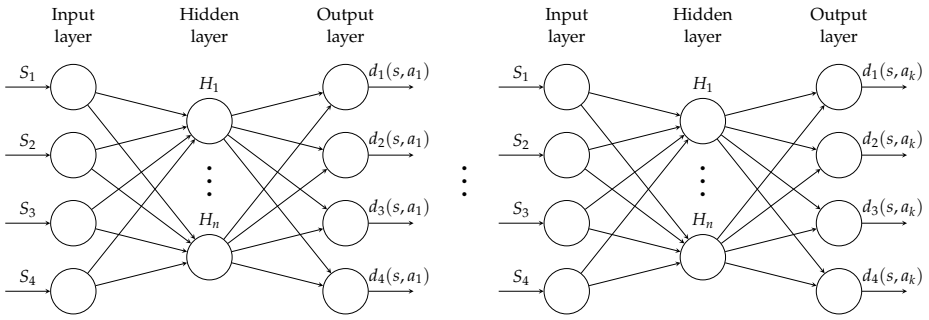


Figure 9.1: Example of an array of ordinal deep neural networks for DQN for reward distribution prediction

1. The Aggregator state at time t is the current neural network weights $D_t = w$.

2. The Aggregator Update: $D_t, s_t, a_t, r_t, s_{t+1}$ updates the weights w stored in D_t with the newly seen reward using backpropagation (see Section 2.2.1)

3. The Get Ratings function computes the Q-Values with the DNN using the current weights $w = D_t$. $V_t(s,a) = DNN_w(a)$

4. Agent.ChooseAction picks the action maximizing the Q-Value: $a_t = \arg\max_a V_t(s(a), a)$ with $s(a)$ being the unique state of a.

9.3.3 Ordinal Deep Q-Network

In this section, we will describe how the ordinal Q-Learning algorithm can be extended to a deep learning algorithm, just like Q-Learning is extended to DQL. Since ordinal rewards are aggregated by a distribution instead of a numerical value, the neural network is adapted to predict distributions instead of Q-values for every possible action. Hence, for one action the network does not predict a 1-dimensional Q-value, but predicts an u-dimensional reward distribution with u being the length of the ordinal scale. Since this distribution has to be computed for each of $|A|$ actions, the adaptation of the Deep Q-Network algorithm to ordinal rewards requires a differently structured neural network. Contrary to the original Deep Q-Network where one network simultaneously predicts one Q-value for each action, the structure of the ordinal DQN consists of an array of k neural networks: One for each action a. In a deep neural network for the prediction of distributions, every output node of the network computes one distribution value $g_{o_i}(s,a)$. The structure of neural networks used for the prediction of distributions can be seen in Figure 9.1.

The prediction of the ordinal reward distributions $G(s,a)$ for all actions can afterwards be normalized to a probability distribution $\hat{F}$ and used in order to compute the value function $V_\pi(s)$ using the Borda technique (see Section 9.2.1). Once the value function and policy have been evaluated, the ordinal variant of the DQN algorithm follows

a similar procedure as ordinal Q-Learning and updates the prediction of the reward distribution for (s, a) by fitting $D_a^{DQN}(s)$ to the target reward distribution:

$$\hat{D}_a^{DQN}(s) = e_{r_o} sa + \gamma D_{\pi^*(s')}^{DQN}(s') \tag{9.6}$$

The main difference in the update step between ordinal Q-Learning (9.5) and ordinal DQN consists of fitting the neural network of action a for input s to the expected reward distribution by backpropagation instead of updating a Q-table entry (s, a). Additional modifications to the ordinal Deep Q-Network in form of experience replay, the split of the target and evaluation network, and the usage of a Double DQN are done in a similar fashion as described with the standard DQN algorithm in Section 2.6.2.

Looking at the implementation of ordinal Deep Q-Network using the OAF, we can reuse Algorithm 8 using the following method implementations:

1. The Aggregator state at time t is the current neural network weights $D_t = w$.

2. The Aggregator Update: $D_t, s_t, a_t, r_t, s_{t+1}$ updates the weights w stored in D_t with the newly seen reward using backpropagation (see Section 2.2.1) with target reward distribution shown in Formula 9.6.

3. Get Ratings first generates the normalized reward distributions G using the network. Then, the different action Borda-Scores are calculated based on G: $V_t(s, a) = \hat{B}_{\hat{F}, A(s)}(a)$ calculates the Borda-Score for each action.

4. Agent.ChooseAction picks the action maximizing the Borda-Score:
   $a_t = \arg\max_a V_t(s(a), a)$ with $s(a)$ being the unique state of a.

9.4 Experiments and Results

In the following, the standard reinforcement algorithms described in Section 2.6.1 and Section 2.6.2 together with the ordinal reinforcement learning algorithms described in Section 9.3 are evaluated and compared in several testing environments.[4]

The results of the comparison between numerical and ordinal algorithms for the CartPole- and Acrobot-environment in terms of score, win rate, and computational time are shown and investigated starting from Section 9.4.2. This comparison is performed based on the averaged results from 10 and respectively 5 independent runs of Q-Learning and Deep Q-Network on the environments.

4 The source code for the implementation of the experiments can be found in `https://github.com/az79nefy/OrdinalRL`.

9.4.1 Experimental Setup

The environments which are used for evaluation are provided by OpenAI Gym,[5] which can be viewed as a unified toolbox for our experiments. All environments expect an action input after every time step and return feedback in form of the newly reached environmental state, the direct reward for the executed action, and the information whether the newly reached state is terminal. The environments that the algorithms were tested on were *CartPole* and *Acrobot*.[6]

Policies of the reinforcement learning algorithms were modified to use ϵ-greedy exploration [37], which encourages early exploration of the state space and increases exploitation of the learned policy over time. In the experiments, the maximum exploitation is reached after half of the total episodes. In order to directly compare the standard and the ordinal variants of reinforcement learning algorithms, the quality of the learned policy and the computational efficiency is investigated across all environments with varying episode numbers. Information about the quality of the learned policy is derived from the sum of rewards over a whole episode (score) or the win rate while the efficiency is measured by real-time processing time. Additionally to the standard variant with unchanged rewards, the performance of standard Q-Learning algorithms is tested with changed rewards in order to simulate the performance in environments where no optimal reward engineering has been performed. It should be noted that the modifications of the rewards are performed under the constraints of the remaining existing reward order, therefore not changing the transformation to the ordinal scale. The change of rewards (CR) from the existing numerical rewards $r \in \{r_1, ..., r_n\}$ is performed for all rewards by the calculation of $r_{CR,i} = \frac{r_i - min(r)}{100}$.

The parameter configuration of the Q-Learning algorithms is learning rate $\alpha = 0.1$ and discount factor $\gamma = 0.9$. The parameter configuration of the Deep Q-Network algorithm is the learning rate $\alpha = 0.0005$ and discount factor $\gamma = 0.9$. As for the network-specific parameters, the *Adam* optimizer is used for the network fitting, the target network is getting replaced every 300 fitting updates, the experience memory size is 200000 and the replay batch size is 64.

9.4.2 Results: Q-Learning

In Figure 9.2 the scores for the CartPole-environment over the course of 400 and 10000 episodes can be seen which were played by an agent using the ordinal (orange) as well as the standard Q-Learning algorithm, with (red) and without (blue) modified rewards. Additionally, the individual dots in this figure represent the scores achieved by the

5 For further information about OpenAI visit `https://gym.openai.com`.

6 Further technical details about the environments CartPole and Acrobot from OpenAI can be found in `https://gym.openai.com/envs/CartPole-v0/` and `https://gym.openai.com/envs/Acrobot-v1/`.

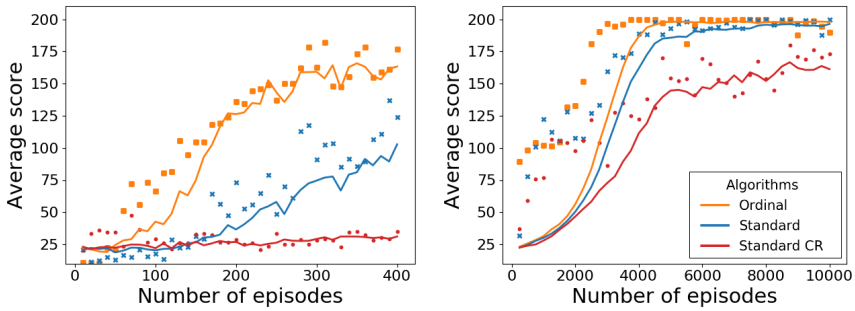


Figure 9.2: CartPole scores of standard and ordinal Q-Learning for 400 and 10000 episodes

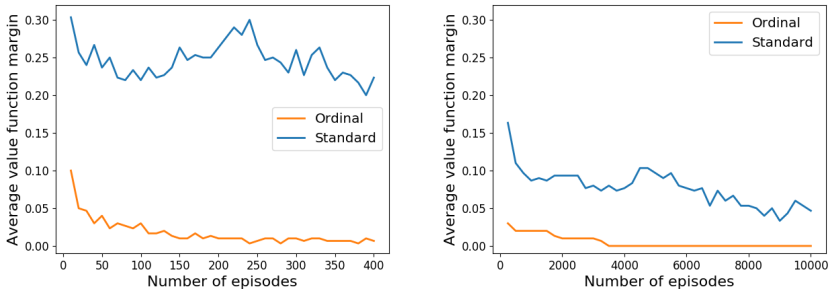


Figure 9.3: Comparison of value function margin for the best action of standard and ordinal Q-Learning for 400 and 10000 episodes of CartPole

respective algorithms by using the optimal policy instead of ϵ-greedy exploration. The evaluation of these scores shows that the ordinal variant of Q-Learning performs better than the standard variant with engineered rewards for 400 episodes and reaches the optimal score of 200 quicker for 10000 episodes. Additionally, the use of ordinal rewards significantly outperforms the standard variant with modified rewards for both episode numbers. Therefore it can be seen that ordinal Q-Learning is able to learn a good policy better than the standard variants for the CartPole-environment.

In order to explain the difference of learned behavior between the standard and ordinal variant, the average relative difference of Q-values $Q(s,a)$ and respectively Borda Scores $\hat{B}_{\hat{F},A(s)}(a)$ for the two possible actions were plotted and compared in Figure 9.3 for standard (blue) and ordinal (orange) Q-Learning. It can be seen for both episode numbers that the policy which is learned by ordinal RL using the Borda technique converges to a difference of 0, meaning that the function $\hat{B}_{\hat{F},A(s)}(a)$ converges to similar values for both actions. This can be interpreted as the policy learning to play safely and rarely entering any critical states where this function would indicate a strong preference towards one action (e.g. in big angles). On the other side, it can be seen for 400 episodes that common RL does not converge towards similar Q-values for the actions over time, and therefore a

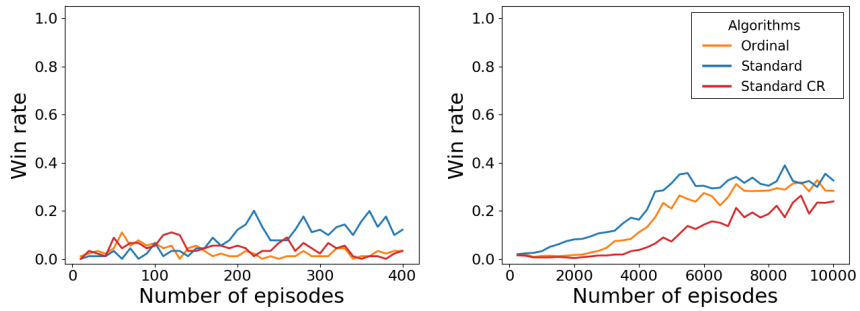


Figure 9.4: Acrobot win rates of standard and ordinal Q-Learning for 400 and 10000 episodes

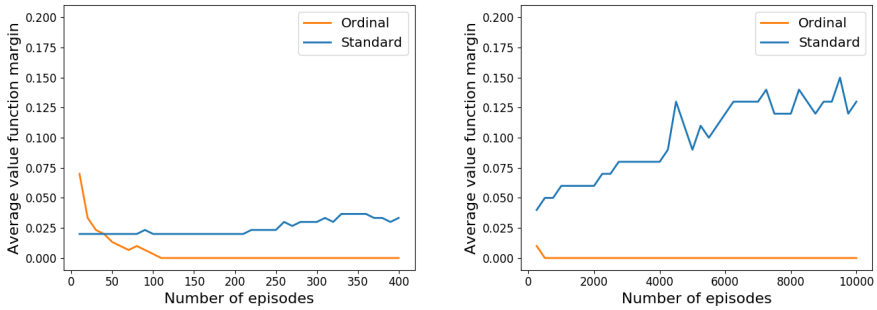


Figure 9.5: Comparison of value function margin: AcrobotComparison of value function margin for the best action of standard and ordinal Q-Learning for 400 and 10000 episodes of Acrobot

policy is learned that enters critical states more often. It should be noted that the Q-value differences for standard Q-Learning converge to 0 for evaluations with more episodes and a safe policy is eventually learned as well.

In Figure 9.4 the win rates from the Acrobot-environment were plotted over the course of 400 and 10000 episodes similarly to the scores for the CartPole-environment and it can be seen for low episode numbers that while the policy learned by the standard variant of Q-Learning with unchanged rewards performs better than the policy learned by the ordinal variant, changing the numerical values of rewards yields the same performance as the ordinal variant. But for high episode numbers, it should be noted that the ordinal variant reaches a similar performance as the standard variant with a win rate of 0.3 after 10000 episodes and clearly outperforms the win rate of the standard Q-Learning algorithm with CR.

Similar to the CartPole-environment, the $\hat{B}$- and Q-function margins of the best actions over the course of 400 and 10000 episodes were compared in Figure 9.5 and yield different observations for the standard and ordinal variants. While the ordinal variant decreases the relative margin of $\hat{B}_{\hat{f},A(s)}(a)$ of the best action and therefore learns a policy which plays safely, the standard variant learns a policy that maximizes the Q-value margin of the best action and therefore follows a policy

Number of	CartPole		Acrobot	
episodes	Standard	Ordinal	Standard	Ordinal
400	2.10 s	4.17 s	35.74 s	52.85 s
2000	10.07 s	24.86 s	174.38 s	266.40 s
10000	67.29 s	130.09 s	855.15 s	1258.30 s
50000	354.52 s	667.87 s	4149.78 s	6178.76 s

Table 9.1: Computation time comparison of standard and ordinal Q-Learning for varying episode numbers

that enters critical states more often. Therefore, it can be concluded that the learned policies differ. While the standard variant learns a good policy quicker, it should be noted that both policies perform comparably well after many episodes despite the policy differences.

As can be seen in Table 9.1, using the ordinal variant results in an additional computational load by a factor between 0.8 and 1.2 for CartPole and 0.5 for Acrobot. The additionally required computational capacity is caused by the computation of the Borda-Score which is less efficient than computing the expected sum of rewards. This factor could be reduced by using the iterative update of the Borda-function as described in [16].

9.4.3 Results: Deep Q-Network

In Figure 9.6, the scores achieved in the CartPole-environment by the ordinal as well as the standard Deep Q-Network, with and without CR, can be seen over the course of 160 and 1000 episodes. For 160 episodes it can be seen that ordinal DQN, as well as the standard variant without CR, converge to a good policy reaching an episode score close to 150. Contrary to this performance, modified rewards negatively impact standard Q-Learning and therefore its performance is significantly worse, not reaching a score above 100. Additionally, for low episode numbers, it should be noted that the policy learned by the ordinal variant of Deep Q-Network is able to achieve good scores faster than the standard variant matching the observation made for the Q-Learning algorithms. The evaluation for 1000 episodes shows that the performances of standard, with and without CR, and ordinal DQNs are comparable.

Figure 9.7 plots the win rate of Deep Q-Network algorithms for the Acrobot-environment over the course of 160 and 1000 episodes. For 160 episodes standard DQN with engineered rewards performs better than the ordinal variant but loses this quality once the rewards are modified. For high episode numbers it can be seen that the ordinal variant is comparable to the standard algorithm without CR and solves the environment with a win rate of close to 1.0, but clearly outperforms the standard DQN with modified rewards which is only

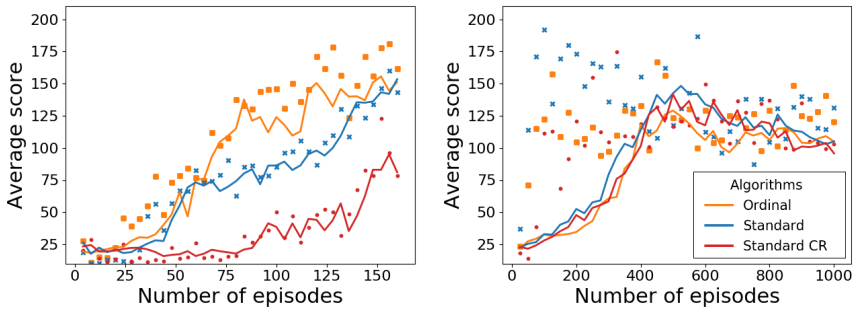


Figure 9.6: CartPole scores of standard and ordinal DQN for 160 and 1000 episodes

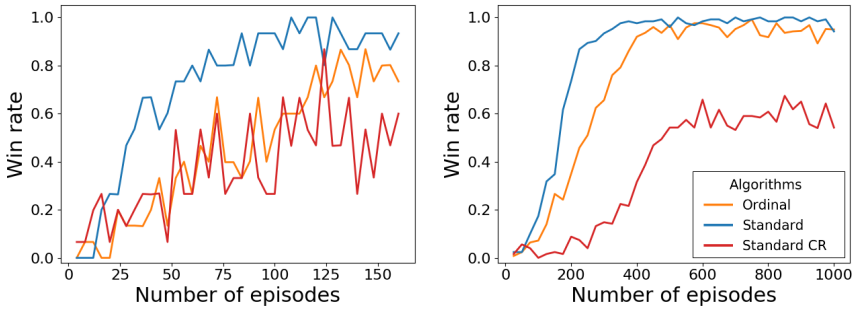


Figure 9.7: Acrobot win rates of standard and ordinal DQN for 160 and 1000 episodes

able to achieve a win rate of 0.6. It should be noted that all variants of DQN are able to learn a better policy than their respective Q-Learning algorithms, achieving a higher win rate in less than 160 episodes.

Additionally, it should be noted that the use of the ordinal variant of DQN adds an additional computational factor between 0 and 0.5 for the CartPole-environment and 1.0 for the Acrobot-environment, as can be seen in Table 9.2.

Since the evaluation of the ordinal Deep Q-Network algorithm shows comparable results to the standard DQN with engineered rewards and furthermore outperforms the standard variant with modified rewards, it can be concluded that the conversion of the Deep Q-Network algorithm to ordinal rewards is successful. Therefore it has been shown that algorithms of deep reinforcement learning can as well be adapted to the use of ordinal rewards.

9.5 Conclusion

In this chapter, we have shown that the use of ordinal rewards for reinforcement learning can reach the quality of standard reinforcement learning algorithms with numerical rewards. Our approach is based on estimations on normalized reward probabilities. Doing so involves limitations as the probabilities do not give information about the number of rewards. Nevertheless, we found evidence that it might

Number of	CartPole		Acrobot	
episodes	Standard	Ordinal	Standard	Ordinal
160	1520.01 s	2232.48 s	3659.44 s	7442.49 s
400	6699.69 s	7001.79 s	9678.80 s	19840.88 s
1000	15428.41 s	15526.84 s	23310.36 s	47755.90 s

Table 9.2: Computation time comparison of standard and ordinal DQN

improve the performance in certain settings. We compared RL algorithms for both numerical and ordinal rewards on a number of tested environments and demonstrated that the performance of the ordinal variant is mostly comparable to the learned common RL algorithms that make use of engineered rewards while being able to significantly improve the performance for modified rewards.

Finally, it should be noted that ordinal reinforcement learning enables the learning of a good policy for environments without much effort to manually shape rewards. We hereby lose the possibility of reward shaping to the same degree that numerical rewards would allow, but therefore gain a more simple-to-design reward structure. Hence, our variant of reinforcement learning with ordinal rewards is especially suitable for environments that do not have a natural semantic of numerical rewards or where reward shaping is difficult. Additionally, this method enables the usage of new and unexplored environments for RL only with the specification of an order of desirability instead of the needed effort of manually engineering numerical rewards with sensible semantic meaning.

10 Conclusion

In Chapter 1, we stated the research questions of this work. We now revisit these challenges and recapitulate on the solutions and contributions we have provided to manage them. 1) Can we develop an MCTS solution that works with ordinal feedback? 2) Which problems do arise and can we circumvent them? 3) Can we transform the solution of 1) to other but similar problem definitions?

10.1 Challenges Revisited

After the foundations that present needed background knowledge like MAB, DB, MDP, Neural Networks, and Reinforcement Learning and common algorithms, we have introduced the first algorithm trying to tackle the first research question: PB-MCTS.

10.1.1 The Preference-Based Approach

The main idea is to take a dueling bandits algorithm and leverage it to tree search, just like MCTS is based upon UCB1. The experiments have shown that PB-MCTS can solve the 8-puzzle with ordinal rewards in a comparable strength to MCTS using real-valued rewards. Looking at research question 1, PB-MCTS is an MCTS algorithm that utilized ordinal feedback, but there are major problems with this approach (leading to research question 2): First, PB-MCTS has worse sampling efficiency than plain MCTS: Due to the dueling bandits approach, it needs a quadratic amount of samples to gain the same information as plain MCTS, since one sample only holds information about one pair of arms. This information does not hold any information about the other arms or how those two arms compare to the other arms. The second big problem is the asymmetric growth of the search tree: One advantage of MCTS is to focus the search on interesting state spaces where MCTS is unsure what the best action is. Hence, there can be parts of the search tree that are much deeper than others. In PB-MCTS this advantage is limited, since (ignoring the case of picking the same arm twice what leads to no information gain) PB-MCTS does not select a path from the root node, but a binary subtree. One path of that tree is the most interesting one and the others are only sampled to gain information about nodes of that tree, which might not even be part of the interesting path. All of those problems do increase with a higher branching factor, and the problem used in the experiments already has a quite low branching factor. To answer the second part

of research question 2 *How to avoid those problems*, we have thought of pruning the selected subtree or similar techniques, but we have eventually chosen a different way leaving PB-MCTS behind: Instead of using preferences of actions and going with the dueling bandits approach, we want to sample ordinal rewards directly and aggregate them in such a way that we can compare one action to another by using all samples of both arms. Furthermore, we can use the voting theory to determine the best action or the next action to sample.

10.1.2 The Ordinal Framework

Since this approach can be formulated without any concept of tree search and looking at research question 3 (*Can we transform the solution of 1) to other but similar problem definitions?*), we have introduced a novel ordinal reward framework in Chapter 5 that is focused around the aggregation of samples and comparison of actions. In later chapters, we have used that framework to create ordinal algorithms for MAB, MCTS, and RL.

10.1.3 Borda Algorithms

Starting with Borda-UCB1, a novel MAB algorithm that uses our framework and is based upon UCB1, we introduce the Borda-Score: A technique to compare different actions given ordinal samples, which is based upon voting theory. We have developed this algorithm without knowledge in parallel with another research group (ref). Since they have managed to publish their version prior to us, we have acknowledged that success but for the sake of this thesis we claim that our algorithms are novel.

Concerning research question 1, we have introduced Borda-MCTS based upon Borda-UCB1 in Chapter 7 and successfully tested it using different environments on the GVGAI Framework. Our experiments have shown that Borda-MCTS can outperform vanilla MCTS and PB-MCTS in these environments.

Adding content to research question 2, a theoretical problem of the used Borda-Score implementation is that the aggregation and comparison of actions need more time the more ordinal rewards are seen. In Chapter 8 we have introduced an online ordinal bucketing algorithm that is able to reduce the number of ordinal rewards.

Finally, we have shown that our ordinal reward framework can be used in other problem settings that MAB and MCTS: In Chapter 9 we used our framework to introduced ordinal algorithm versions of Q-Learning and Deep-Q-Learning and evaluated them using the Open AI Gym.

10.2 Future Work

In this last section, we give insight about possible future work based upon this work.

In Chapter 6, we have introduced the Borda winner as one possibility to define a winner of an election process and showed how a bandit algorithm can identify a Borda winner, based on the upper confidence bound approach. As UCB algorithms are popular and come with theoretical background, they are not the only option to solve bandit problems. For example, one could use Thompson sampling to solve a bandit problem and identify the Borda winner this way. There are also other alternatives, like the Condorcet winner, that could be analyzed in comparison to the Borda approach we have tackled in this thesis. In fact, a recent paper has proposed a Thompson sampling bandit algorithm to identify the Borda winner [47]. As we have used Borda-UCB as a basis to build Borda-MCTS on, a natural next step could be building Thompson-Borda-MCTS algorithms or variations playing the Condorcet winner.

Throughout our experiments, we have found several cases where we have treated numerical rewards as ordinal rewards and compared UCB1 with Borda-UCB or UCT with Borda-MCTS. In some of those cases, the Borda variants beat the numerical alternatives significantly. We were not able to identify a root cause for that or to give a statement of when ordinal rewards will outperform numerical rewards, but these might exist and that is an interesting path to follow and explore.

In the last chapter, we have created a RL algorithm that works on ordinal rewards. We have followed the approach of estimating the probability for seeing rewards and calculating the Borda winner based on this data. This approach can solve the baseline problems as we have shown, but the definition of optimality may not always be intuitive, as one very good reward will always be preferred over any number of worse rewards. As the absence of numerical trade-offs like *a cured patient permits one patient do die* can be a good thing, our approach does not support any trade-offs, even like *three coins are worth one diamond*, which we might want to consider. Hence, an interesting concept to investigate is how to implement certain trade-offs in ordinal algorithms like Borda-UCB.

Bibliography

[1] Nir Ailon, Zohar Karnin, and Thorsten Joachims. "Reducing dueling bandits to cardinal bandits." In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR. 2014, pp. 856–864.

[2] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. "Concrete problems in AI safety." In: *arXiv preprint arXiv:1606.06565* (2016).

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem." In: *Machine learning* 47.2 (2002), pp. 235–256.

[4] Yael Ben-Haim and Elad Tom-Tov. "A streaming parallel decision tree algorithm." In: *Journal of Machine Learning Research* 11.2 (2010).

[5] Duncan Black. "Partial justification of the Borda count." In: *Public Choice* 28.1 (1976), pp. 1–15.

[6] Duncan Black. *The Theory of Committees and Elections*. Springer, 1958.

[7] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. "A survey of Monte Carlo tree search methods." In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[8] Róbert Busa-Fekete and Eyke Hüllermeier. "A survey of preference-based online learning with bandit algorithms." In: *Proceedings of the International Conference on Algorithmic Learning Theory (ALT)*. Springer. 2014, pp. 18–39.

[9] Marie JAN de Caritat and Marquis De Condorcet. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. L'imprimerie royale, 1785.

[10] Janez Demšar. "Statistical comparisons of classifiers over multiple data sets." In: *The Journal of Machine Learning Research* 7 (2006), pp. 1–30.

[11] Hilmar Finnsson. "Simulation-based general game playing." PhD thesis. School of Computer Science, Reykjavík University, 2012.

[12] Hugo Gilbert and Paul Weng. "Quantile reinforcement learning." In: *Proceedings of the Workshop on Machine Learning Research (MLR)*. 2016.

[13] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep Learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[14] Hosagrahar Visvesvaraya Jagadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Kenneth C Sevcik, and Torsten Suel. "Optimal histograms with quality guarantees." In: *Proceedings of the International Conference of Very Large Data Bases (VLDB)*. Vol. 98. 1998, pp. 24–27.

[15] Kevin G Jamieson, Sumeet Katariya, Atul Deshpande, and Robert D Nowak. "Sparse dueling bandits." In: *Proceedings of the Artificial Intelligence and Statistics (AISTATS)*. PMLR. 2015, pp. 416–424.

[16] Tobias Joppen and Johannes Fürnkranz. "Ordinal Monte Carlo tree search." In: *Proceedings of the AIII IJCAI Workshop on Monte Carlo Search (MCS)*. 2020.

[17] Tobias Joppen and Johannes Fürnkranz. "Ordinal Monte-Carlo tree search." In: *arXiv preprint arXiv:2101.10670* (2020). Abandoned preprint.

[18] Tobias Joppen, Miriam U Schröder, Nils Schröder, Christian Wirth, and Johannes Fürnkranz. "Informed hybrid game tree search for general video game playing." In: *IEEE Transactions on Games* 10.1 (Mar. 2018), pp. 78–90.

[19] Tobias Joppen, Tilman Strübig, and Johannes Fürnkranz. "Ordinal bucketing for game trees using dynamic quantile approximation." In: *Proceedings of the IEEE Conference on Games (CoG)*. IEEE. 2019, pp. 1–8.

[20] Tobias Joppen, Christian Wirth, and Johannes Fürnkranz. "Preference-based Monte Carlo tree search." In: *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI)*. Springer. 2018, pp. 327–340.

[21] Ahmed Khalifa, Aaron Isaksen, Julian Togelius, and Andy Nealen. "Modifying MCTS for human-like general video game playing." In: *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*. Vol. 25. 2016, pp. 2514–2520.

[22] Joshua D Knowles, Richard A Watson, and David W Corne. "Reducing local optima in single-objective problems by multi-objectivization." In: *Proceedings of the International Conference on Evolutionary Multi-Criterion Optimization (EMO)*. Springer. 2001, pp. 269–283.

[23] Levente Kocsis and Csaba Szepesvári. "Bandit based Monte-Carlo planning." In: *Proceedings of the European Conference on Machine Learning (ECML)*. Springer. 2006, pp. 282–293.

[24] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. "The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments." In: *IEEE Transactions on Computational Intelligence and AI in games* 1.1 (2009), pp. 73–89.

[25] Long-Ji Lin. "Reinforcement Learning for Robots Using Neural Networks." PhD thesis. Carnegie Mellon University, 1992.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A Riedmiller. "Playing Atari with deep reinforcement learning." In: *CoRR* abs/1312.5602 (2013).

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin A Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (2015), pp. 529–533.

[28] Andrew Y Ng, Daishi Harada, and Stuart Russell. "Policy invariance under reward transformations: Theory and application to reward shaping." In: *Proceedings of the International Conference on Machine Learning (ICML)*. Vol. 99. 1999, pp. 278–287.

[29] Tom Pepels, Mark HM Winands, and Marc Lanctot. "Real-time Monte Carlo tree search in ms pac-man." In: *IEEE Transactions on Computational Intelligence and AI in games* 6.3 (2014), pp. 245–257.

[30] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. "General video game AI: A multitrack framework for evaluating agents, games, and content generation algorithms." In: *IEEE Transactions on Games* 11.3 (2019), pp. 195–214.

[31] Diego Perez-Liebana, Sanaz Mostaghim, and Simon M Lucas. "Multi-objective tree search approaches for general video game playing." In: *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2016, pp. 624–631.

[32] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon Lucas. "General video game AI: Competition, challenges and opportunities." In: *Proceedings of the AAAI Conference on Artificial Intelligence (AI)*. Vol. 30. 1. 2016.

[33] Marc Ponsen, Geert Gerritsen, and Guillaume Chaslot. "Integrating opponent models with Monte-Carlo tree search in poker." In: *Proceedings of the AAAI Conference on Interactive Decision Theory and Game Theory*. Vol. 3. 2010, pp. 37–42.

[34] Siddartha Y Ramamohan, Arun Rajkumar, and Shivani Agarwal. "Dueling bandits: Beyond Condorcet winners to general tournament solutions." In: *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*. 2016, pp. 1253–1261.

[35] Monia Ranalli and Roberto Rocci. "Clustering methods for ordinal data: A comparison between standard and new approaches." In: *Advances in Statistical Models for Data Analysis*. Springer, 2015, pp. 221–229.

[36] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. "Mastering the game of Go without human knowledge." In: *Nature* 550.7676 (2017), pp. 354–359.

[37] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[38] Louis L Thurstone. "A law of comparative judgment." In: *Psychological review* 34.4 (1927), p. 273.

[39] Tanguy Urvoy, Fabrice Clerot, Raphael Féraud, and Sami Naamane. "Generic exploration and k-armed voting bandits." In: *Proceedings of the International Conference on Machine Learning (ICML)*. 2013, pp. 91–99.

[40] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." In: *Proceedings of the AAAI Conference on Artificial Intelligence (AI)*. Vol. 30. 1. 2016.

[41] András Vargha and Harold D Delaney. "A critique and improvement of the CL common language effect size statistics of McGraw and Wong." In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[42] Christopher JCH Watkins and Peter Dayan. "Q-learning." In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[43] Paul Weng. "Markov decision processes with ordinal rewards: Reference point-based preferences." In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Vol. 21. 1. 2011.

[44] Paul Weng. "Ordinal decision models for Markov decision processes." In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Vol. 20. 2012, pp. 828–833.

[45] Paul Weng, Róbert Busa-Fekete, and Eyke Hüllermeier. "Interactive q-learning with ordinal rewards and unreliable tutor." In: *Proceedings of the ECML/PKDD-13 Workshop on Reinforcement Learning from Generalized Feedback: Beyond Numeric Rewards*. 2013.

[46] Christian Wirth, Riad Akrour, Gerhard Neumann, and Johannes Fürnkranz. "A survey of preference-based reinforcement learning methods." In: *Journal of Machine Learning Research* 18.136 (2017), pp. 1–46.

[47] Liyuan Xu, Junya Honda, and Masashi Sugiyama. "Dueling bandits with qualitative feedback." In: *Proceedings of the AAAI Conference on Artificial Intelligence (AI)*. Vol. 33. 01. 2019, pp. 5549–5556.

[48] Georgios N Yannakakis, Roddy Cowie, and Carlos Busso. "The ordinal nature of emotions." In: *Proceedings of the International Conference on Affective Computing and Intelligent Interaction (ACII)*. Vol. 7. IEEE. 2017, pp. 248–255.

[49] H Peyton Young. "Condorcet's theory of voting." In: *The American Political Science Review* (1988), pp. 1231–1244.

[50] Yisong Yue, Josef Broder, Robert Kleinberg, and Thorsten Joachims. "The k-armed dueling bandits problem." In: *Journal of Computer and System Sciences* 78.5 (2012), pp. 1538–1556.

[51] Yisong Yue and Thorsten Joachims. "Interactively optimizing information retrieval systems as a dueling bandits problem." In: *Proceedings of the International Conference on Machine Learning (ICML)*. Vol. 26. 2009, pp. 1201–1208.

[52] Alexander Zap. "Ordinal Reinforcement Learning." MA thesis. Technische Universität Darmstadt, 2019.

[53] Alexander Zap, Tobias Joppen, and Johannes Fürnkranz. "Deep ordinal reinforcement learning." In: *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*. Springer. 2019, pp. 3–18.

[54] Masrour Zoghi, Shimon Whiteson, Remi Munos, and Maarten Rijke. "Relative upper confidence bound for the k-armed dueling bandit problem." In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR. 2014, pp. 10–18.

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 02.02.2021

Tobias Joppen