Ordinal Bucketing for Game Trees using Dynamic Quantile Approximation

Tobias Joppen
Knowledge Engineering Group
TU Darmstadt
Darmstadt, Germany
tjoppen@ke.tu-darmstadt.de

Tilman Strübig
TU Darmstadt
Darmstadt, Germany
tilman.struebig@googlemail.com

Johannes Fürnkranz
Knowledge Engineering Group
TU Darmstadt
Darmstadt, Germany
juffi@ke.tu-darmstadt.de

***Abstract*—In this paper, we present a simple and cheap ordinal bucketing algorithm that approximately generates q-quantiles from an incremental data stream. The bucketing is done dynamically in the sense that the amount of buckets q increases with the number of seen samples. We show how this can be used in Ordinal Monte Carlo Tree Search (OMCTS) to yield better bounds on time and space complexity, especially in the presence of noisy rewards. Besides complexity analysis and quality tests of quantiles, we evaluate our method using OMCTS in the General Video Game Framework (GVGAI). Our results demonstrate its dominance over vanilla Monte Carlo Tree Search in the presence of noise, where OMCTS without bucketing has a very bad time and space complexity.**

***Index Terms*—bucketing, ordinal, rewards, MCTS, GVGAI, general game playing, quantiles**

I. Introduction

Ordinal data are widely used in many real-world scenarios such as in ratings or questionnaires. In many cases, the set of possible values is limited to a low number of ordinal values, such as 1 to 5 stars, but this is not necessary the case. The basic assumption of ordinal data is that nothing but the ordering of the values is known. In particular, no distance measure can be assumed. This implies, e.g., that averaging, adding or multiplying data values is impossible, in contrast to common real-valued data. Therefore, ordinal data are much more difficult to handle than real-valued data, which is one reason why they are often interpreted as numerical values.

For example, the framework for the General Video Game AI (GVGAI) competitions, which we use in this paper for evaluation purposes, includes a large variety of games that can be played. For doing so, it provides game playing agents with a numerical score for the given state of a game. Increasing the score often correlates to performing well and approaching the goal. Examples for actions that lead to an increase in score include collecting diamonds, catching or slaying enemies, solving a minor puzzle or detecting a key for a door. With few exceptions and across all games, such events increase the score by exactly one point. It is likely, that this is not a meaningful distance measure but only an indication of success, or a number to derive an ordering or preferences over states. Hence, there are arguments to view those scores as ordinal values.

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

Monte Carlo Tree Search, a basic algorithm often used in GVGAI agents, interprets these scores as real-valued feedback [2], [5]. In previous work, we have proposed Ordinal-MCTS (OMCTS), an MCTS variant that treats these scores in an ordinal fashion [4]. The OMCTS algorithm has a linear factor to time and space complexity dependent on the number of ordinal rewards seen. This is sufficient for domains with a low number of possible ordinal values, but may become excessive in comparison to MCTS once this number rises.

In this paper, we present an algorithm that uses bucketing for bounding the number of ordinal values to make OMCTS work efficient with any stream of ordinal values. This is a problem especially in settings with noisy reward signals. We investigate this setting by applying artificial noise to GVGAI games. Due to the fact that OMCTS spans a tree of game states, and uses bucketing in each of those states, it should be fast and performant for any amount of data with as little overhead as possible.

One property of MCTS is the *asymmetric growth* of its search tree. Actions that lead to better states are visited and explored more often. Hence, MCTS spends more time searching for good solutions and less time in less interesting parts of a game tree. Similarly, one does not want to spend a lot of time or overhead to bucket the ordinal rewards in bad states. Instead, one is fine with having a coarser approximation of the seen rewards for non-optimal states. This is in contrast to the well explored parts of the search tree: Here, one wants to have a fine-grained bucketing to be able to identify the very best action, and thus one is willing to put more time for creating this bucketing. Since we do not know a priori whether a given action needs a fine-grained bucketing or a rough approximation, we are in need for a dynamic method that improves its quality the more data is seen. For example, compare the root node with a newly expanded node: The root node is frequently visited and needs very detailed information about its reward distribution whereas a node that has just been generated does not need any bucketing at all.

In the following chapter, we start with a discussion of related work that also focuses on reducing the size of a set by merging values or identifying meta concepts, and relate ordinal bucketing to quantile approximation.

II. 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 [3]. They can often can 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 [1], [3]. In this paper, we take a look at a 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 [8].

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. Looking once more at the root and leaf node example, the root node might want to have its action rewards organized in many quantiles whereas the leaf node is 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. In comparison to other bucketing or quantile approximating algorithms, our approach only stores very little information and often needs to resort to random decisions, which nevertheless leads to good results. We will 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 to analyze the influence of OMCTS using this approach. As a baseline we also compare to vanilla MCTS.

III. Searching with Ordinal Rewards

A. Ordinal Markov Decision Processes

Markov Decision Processes (MDPs) [7] are problems in which an agent has to repeatedly choose one action $a \in A$ from a given set of possibilities. Once an action is chosen, the agent moves to another state $s \in S$ while receiving a short-term reward $r \in R$. This other state may be a terminal state (like the end of a game) or has a new set of actions $A(s) \subseteq A$ to choose from.. The agent's task is to find a good policy to maximize some quality measure. The most prominent measure of success is the *cumulative regret*, i.e., the difference between the sampled and the optimal (expected) reward.

Ordinal MDPs [9] are a variant of this setting, in which the agent observes an ordinal reward signal $o \in Q$ instead of a real-valued reward. To make sense of ordinal rewards, an qualitative scale $E = \{o_1 \succ o_2, \dots \succ o_n\}$ over all $o_i \in Q$ is given. Since no metric is applicable in Q, and therefore rewards can not be trivially aggregated, one has to use other quality measures for OMDPs.

B. Ordinal Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a popular algorithm to approximately solve MDPs in real time [2]. The algorithm iteratively builds up a growing model of the game tree. One iteration exists of four phases: The *selection step* starts at the root node of the tree and iteratively chooses one action given historical information about those actions. This is often done using the *UCT* formula [6], which trades off actions that are perceived as good (exploitation) and actions that have not been visited often (exploration). Once the selection reaches a leaf node, the *expansion step* adds one or more child nodes to the tree. From there on, a *simulation* is started which performs random actions until a terminal node is found or a computational limit is reached. In the last step, the (heuristic) value of this final state is used to update the information of all actions along the chosen path, which may change the selected actions in the next iteration.

In previous work, we have introduced an variant of this algorithm for OMDPS, in which the values obtained at the end of each iteration are on an ordinal scale [4]. Its key ingredient is a relative dominance measure to rate actions for OMDPs. Here, an action is rated in comparison to its alternatives:

$$B(a) = \frac{1}{|A|-1} \sum_{b;b \neq a}^{A} \Pr[a \succ b], \tag{1}$$

where A is the set of possible actions, $a \in A$ is the action to inspect and $\Pr[a \succ b]$ is the tie-normalized chance of a beating b given two random samples from those arms. This probability can be estimated empirically. Current methods have time and space complexities that are linear in the size of Q, the set of reward signals [4]. Once this set grows out of decent bounds or maybe even becomes infinite (e.g. in the presence of noise) the estimation of B becomes too costly. The method proposed in this paper can bound the complexity by a fixed or logarithmic growing maximum number q of buckets, so that an obtained reward does not have to be compared to all previously observed values, but only to $q-1$ points given by the observed q-quantiles.

OMCTS uses (1) as the exploitation term in a modified UCT formula:

$$a^* = \arg\max_{a \in A} B(a) + 2C\sqrt{\frac{2 \ln n}{n_a}}, \tag{2}$$

where n is the number of actions played, n_a is the number of action a played so far and a^* is the next action to choose in the selection step.

The next section introduces three ordinal bucketing methods. In Section VI, we will show how these can be integrated into OMCTS.

IV. 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.

A. 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 have a need for 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 $g_n \in \mathbb{N}$ 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 value 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 for bucketing, if one wants to calculate the 2-quantile (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.

B. 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 g_o, $U(o)$ is its upper bound and $D(o)$ are the data stored in g_o. The number of stored elements $N(o)$ of g_o is updated by calling '*Store o*'.

1) First-n-Bucketing: A simple algorithm for bucketing ordinal values takes the first n distinct ordinal values and uses them as upper bounds for its buckets. 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 1). 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.

Algorithm 1 Adding a value with **First-n-Bucketing**

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

2) k-Log-Growing: The next idea addresses the dynamically increasing number of buckets. Here, the number of buckets have 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 2).

In the initialization phase, an empty bucket spanning the complete range is added. At the beginning, new samples are added to this one bucket. Once the upper bound of available buckets increases, a new bucket can be added. Instead of adding a new empty bucket, we split an existing bucket using a pivot point.

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 a 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. 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 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.

Algorithm 2 Adding an ordinal value with k-Log-Growing

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

3) 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 3), boosts the accuracy for few observed values, while it is still able to handle large amounts of data.

Algorithm 3 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 largest bucket
  end if

If we take a look at the space-complexity of our presented algorithms (see Fig. 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 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 $\mathcal{O}((\log t)^2)$.

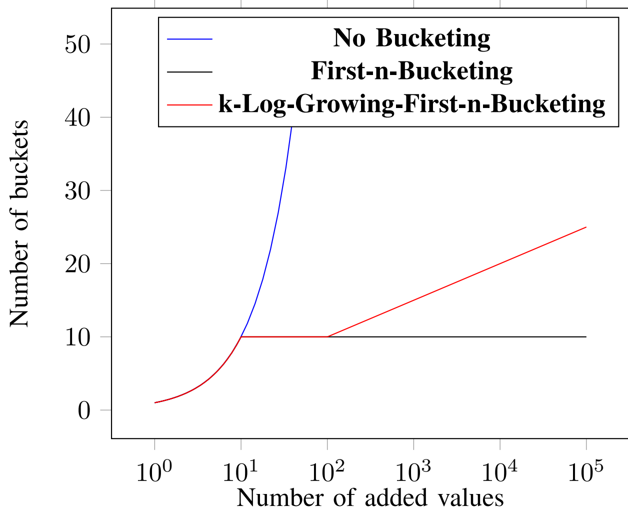

Fig. 1. Space-complexity for no bucketing, First-n-Bucketing and k-Log-Growing-First-n-Bucketing

V. Analysis of Bucketing Error

A. 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 are known. Hence, it is impossible to measure common error terms like the *Sum of Squared Errors*, a common bucketing 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. 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. 4). A value of $n = 5$ is used in the upcoming tests. The next experiment examines the behavior of **k-Log-Growing-First-n-Bucketing** with different values of k for an increasing number of samples (see Fig. 5), followed by a comparison of our three bucketing algorithms, also dependent on the number of samples. Utilizing the results of the previous experiments, we decided to compare the following configurations: First-n-Bucketing with $n = 5$, k-Log-Growing-Bucketing with $k = 2$ and k-Log-Growing-First-n-Bucketing with $n = 5$ and $k = 2$ (see Fig. 6). The last experiment uses different distributions to check the performance for different use-cases, like an exponential falling curve, a gaussian and custom defined distribution (see Fig. 2). 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. 7).

B. Results

In these experiments a single bucketing is used to bucket a bigger stream of data (up to 10^5). Figure 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

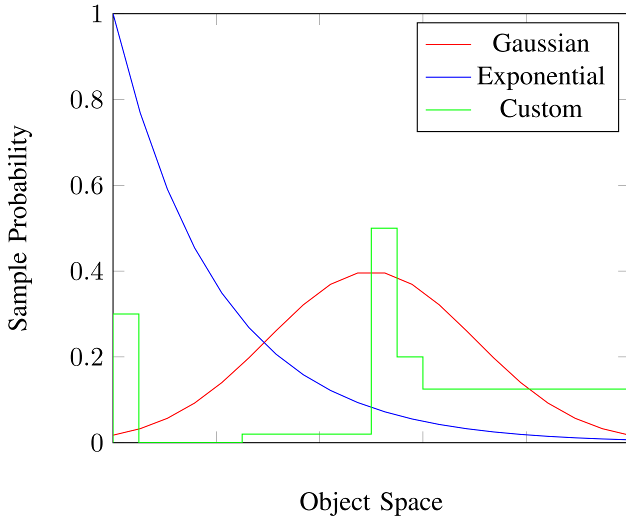

Fig. 2. The three distributions, used in the experiments.

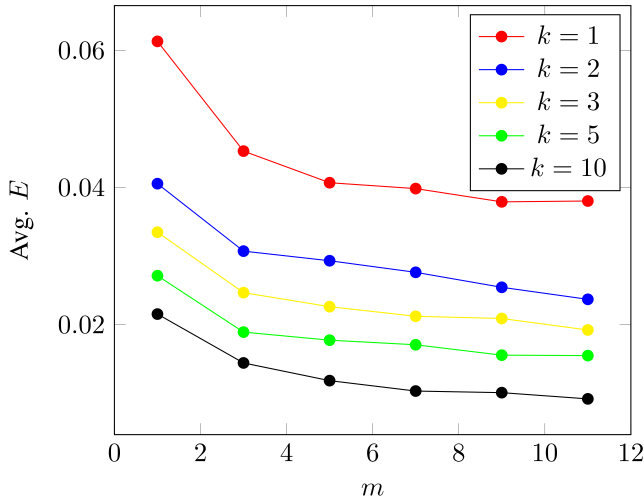

Fig. 3. Average distance to the true percentiles for **k-Log-Growing-Bucketing** with different values of k, dependent on m. Results for 1000 added values, averaged over 100 runs.

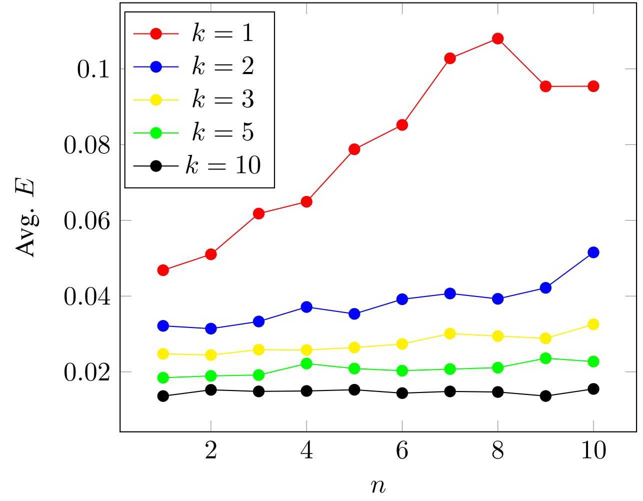

Fig. 4. Average distance to the true percentiles for **k-Log-Growing-First-n-Bucketing** with different values of k, dependent on n. Results for 1000 added values, averaged over 100 runs.

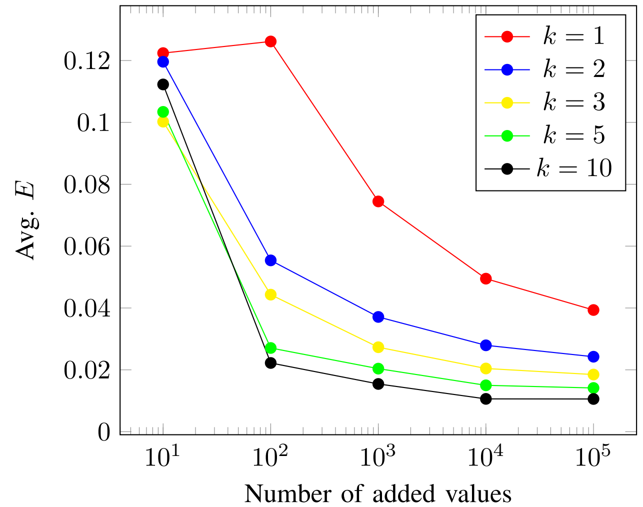

Fig. 5. Average distance to the true percentiles for **k-Log-Growing-First-n-Bucketing** with different values of k, dependent on the number of added values. Results for $n = 5$, averaged over 100 runs.

explained in the previous section, the following tests are done using $m = 3$.

Figure 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 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 3, 4 and 5). Since this behavior was expected, it also justifies the error-measure itself. Figure 5 also shows a fairly stable trend, once the 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 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 performance for large amounts of values, but k-Log-Growing-Bucketing generates a strictly lower error, which also matches the results in Figure 4, where no error-decrease could be seen with an

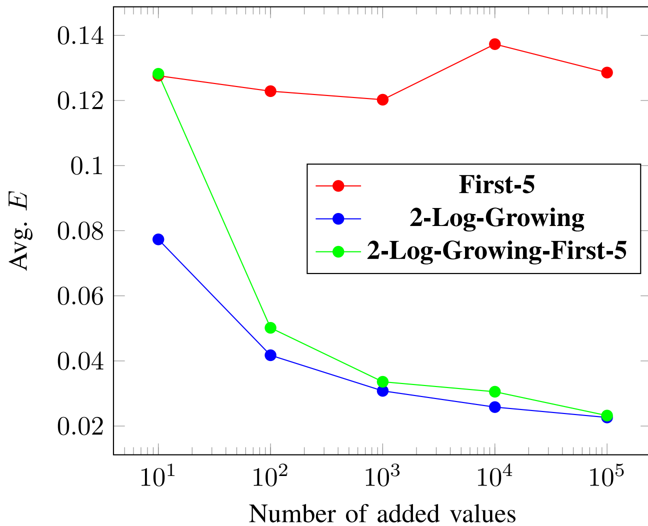

Fig. 6. Average distance to the true percentiles for instances of the three presented algorithms, dependent on the number of added values. Results averaged over 100 runs.

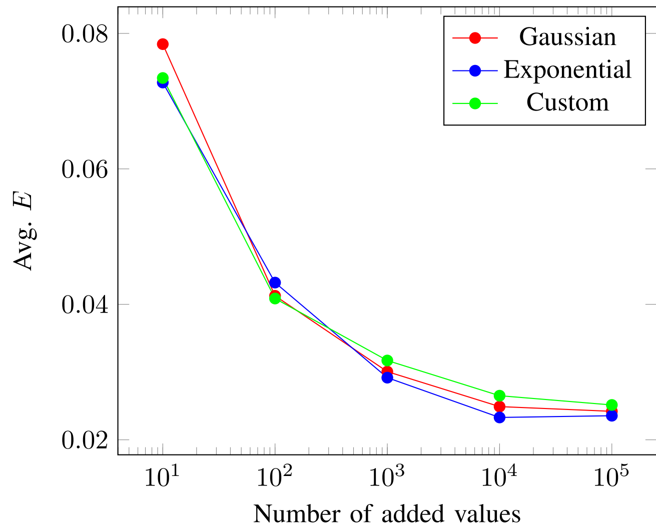

Fig. 7. Average distance to the true percentiles for three different distribution, dependent on the number of added values. Results for **k-Log-Growing-First-n-Bucketing** with $k = 2$ and $n = 5$, averaged over 100 runs.

increasing n.

Finally, Figure 7 exhibits an almost identical performance of k-Log-Growing-First-n-Bucketing on all three used distributions, indicating a good robustness.

VI. Integration of Bucketing into 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

$$\Pr(a_i \succ a_j) = \Pr(a_i > a_i) + \frac{1}{2}\Pr(a_i = a_i). \qquad (3)$$

Hereby, $\Pr(a_i > a_i)$ can be estimated by computing the integral over the sampled Q values using f of a_i and a_j [4]. The linear complexity of OMCTS arises from solving this integral and updating f on all sampled values. Using our bucketing method, we can reduce this complexity. Instead of storing and iterating over the complete list of seen 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.

VII. Experiments on GVGAI Games

A. 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 average number of iterations. The win rate is the most important measure, since the very first task of an 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.

The tested algorithms are MCTS, OMCTS, and OMCTS with the three different bucketing methods introduced in Section IV: 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:

- *Whackamole* is a quite simple game where one walks around and collects mushrooms that randomly spawn and grant score. There also is a cat one has to avoid since

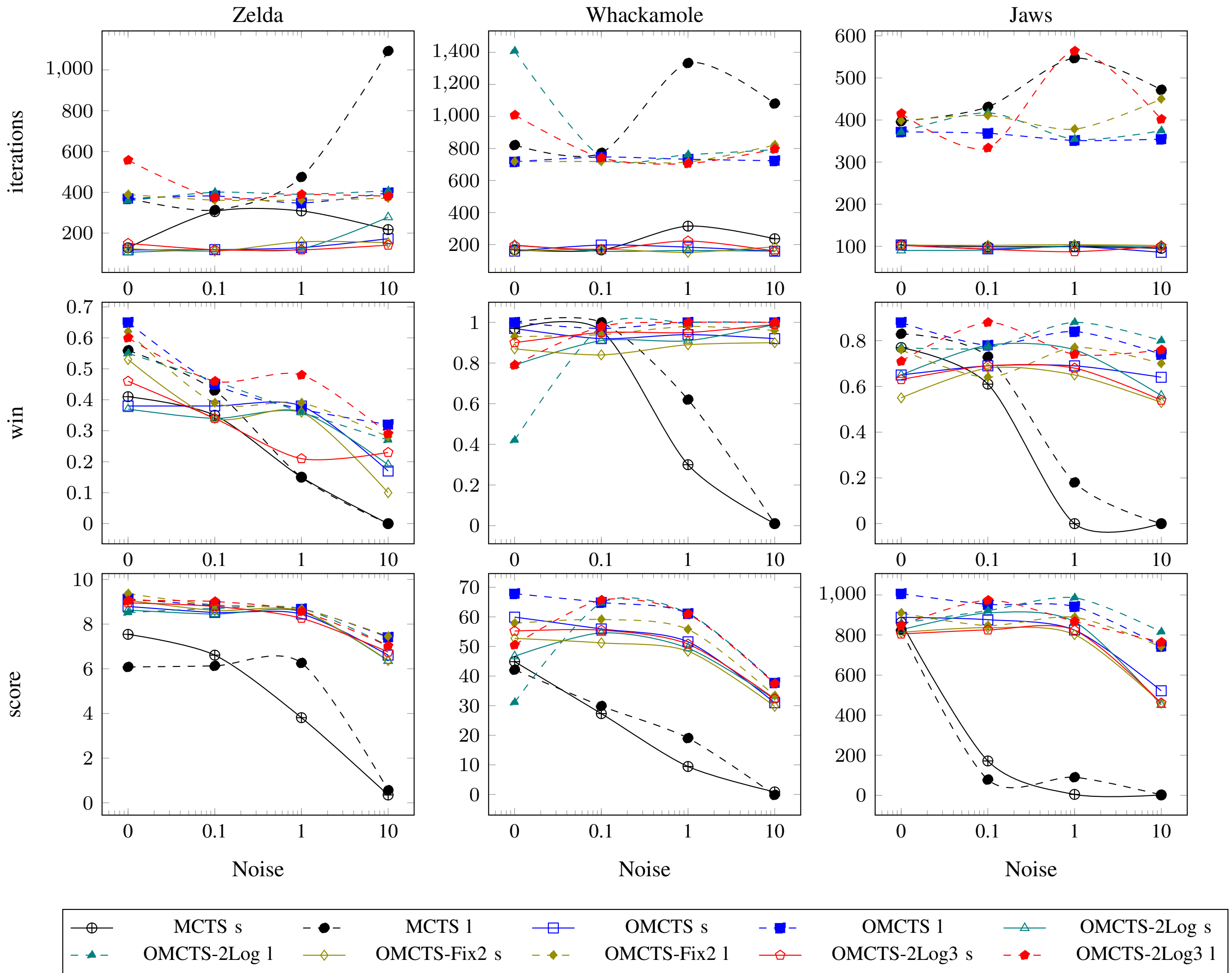

Fig. 8. The results, scores and iterations of all five algorithms on three games having 20ms (*s* postfix) and 400ms (*l* postfix)

touching it is the only way to lose the game. It is easy to avoid this collision.

- *Jaws* has many enemies that can kill the agent, but done right one can kill them easily and eventually reach a +1000 score bonus.
- *Zelda*, unlike the other games, does not have many score changes while playing the game. The player can collect a key (+1 points), and can also slay enemies with your sword that kill you on collision (+2 points for each killed enemy). Interestingly, picking up the key is necessary to win the game whereas killing the enemies is not. Having a lot of noise hence might lead to not picking up the key but slaying the enemies, which can result in a lost game with relatively high score.

  Finally after 2000 game ticks, *Zelda* is lost if the agent did not manage to collect the key and exit through the door.

B. Results

Figure 8 shows the average number of iterations per turn and the average win and score values per game in dependence of different noise levels. Over all games, one can see that the number of wins, the score and the number of iterations do not differ significantly for different bucketing versions. Even using no bucketing does not show a significant difference.

The most outstanding result is how the real-valued MCTS and ordinal MCTS versions behave in the presence of noise: The performance of MCTS constantly drops when the noise is increased having nearly zero wins at a 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 amount 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 IV, 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 chances of seeing the same reward twice goes to zero.

Looking at the average number of iterations, one can see a direct correlation of iterations and lost games. Sadly the number of iterations that the agent can perform seem 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 is 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. Further more, 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 which only is significant for very low stored samples.

VIII. Conclusion

We have proposed an ordinal bucketing method which 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 is able to keep the amount of won games.

Acknowledgment

We gratefully acknowledge the use of the Lichtenberg high performance computer of the TU Darmstadt for our experiments. This work has been supported by the German Research Foundation (DFG).

References

[1] Ben-Haim, Y. and Tom-Tov, E., 2010. A streaming parallel decision tree algorithm. Journal of Machine Learning Research, 11(Feb), pp.849-872.

[2] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S., 2012. A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games, 4(1), pp.1-43.

[3] Jagadish, H.V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K.C. and Suel, T., 1998. Optimal histograms with quality guarantees. In Proc. 24th International Conference on Very Large Data Bases (VLDB-98), pp. 275–286.

[4] Joppen, T. and Fürnkranz, J., 2019. Ordinal Monte Carlo Tree Search. arXiv preprint arXiv:1901.04274.

[5] Joppen, T., Moneke, M.U., Schröder, N., Wirth, C. and Fürnkranz, J., 2018. Informed hybrid game tree search for general video game playing. IEEE Transactions on Games, 10(1), pp.78–90.

[6] Kocsis, L., Szepesvári, C., 2006. Bandit Based Monte-Carlo Planning. Proc. 17th European Conference on Machine Learning (ECML-06), pp. 282–293.

[7] Puterman, M., 1994. Markov Decision Processes: Discrete Stochastic Dynamic Programming. 1st edition. John Wiley & Sons, Inc. New York, NY, USA.

[8] Ranalli, M. and Rocci, R., 2015. Clustering Methods for Ordinal Data: A Comparison Between Standard and New Approaches. In Advances in Statistical Models for Data Analysis, pp. 221-229. Springer.

[9] Weng, P., 2011. Markov decision processes with ordinal rewards: Reference point-based preferences. In Proc. 21st International Conference on Automated Planning and Scheduling (ICAPS-11), Freiburg, Germany.