Efficient B-Tree Insertions Using Proximal Policy Optimization and Hierarchical Attention Models

Alexander Kastius [*1] **Nick Lechtenbörger** [*1] **Felix Schulz** [*1] **Johann Schulze Tast** [*1] **Rainer Schlosser** [1] **Ralf Herbrich** [1]

Abstract

B-trees are a fundamental component of any large database management system. They can grow to noticeable sizes, but their handling is non-trivial. We present a novel approach to use attention-based models with weight sharing across the hierarchical structure of the tree to parse such large trees fast and without the need for excessive training on large clusters. We present a use case in which the model is used in conjunction with PPO to manage write operations on such a tree.

1. Introduction

B-trees are a well-known data structure in the database research community. They are commonly used to index data and make it searchable efficiently. In some practical applications, those trees can become very large, containing millions of entries. If insert and delete operations occur at high frequency throughout the tree's existence, they lead to regular changes to the tree's structure. Those become crucial if, throughout the insertion, a node overflows and needs to be split or is underfilled, in which case they need to be merged. Depending on the tree implementation, noticeable overall performance improvements could be achieved by collecting received operations and reordering them to minimize the overall cost of execution. We propose to learn a representation of the tree that can be efficiently computed during updates and use this to automatically train an agent to choose the order of execution. This approach aims to be scalable without requiring more computational capacity than for small datasets. Several mechanisms, including caching and weight sharing, do assist in achieving this goal.

A B-tree forms a search tree with nodes of variable size. It is characterized by the maximum number of referenced child nodes within a single node, a number called b throughout this paper. To avoid large unfilled nodes, the nodes have a minimum width, which is set to $\lceil 0.5b \rceil$. To stay within these boundaries, inserts and deletes cause reordering of nodes if they are exceeded after insertion or deletion. An illustration of the insert- and delete process with overflowing and underfull nodes is given in Figure 1 and 2 in the Appendix.

Contribution To circumvent the challenges above, we seek to design a network architecture with an efficient caching mechanism to minimize the number of reads and writes on the data structure while parsing and throughout operation. It includes a novel architecture, as well as several additions to improve its efficiency. Throughout the next sections, we show that an agent can be trained based only on interaction with the system and that its policy outperforms other plausible heuristics that could be applied to the same problem. At the same time, once trained and under careful optimization to minimize necessary tree reads, it is quick enough not to diminish the performance improvements it achieves during read/write operations.

2. Related Work

Our literature review covers two aspects: one section refers to several improvements on B-trees, related to our suggested solution; the second focuses on related model setups.

[*]Equal contribution [1]Hasso-Plattner-Institute, University of Potsdam, Potsdam, Germany. Correspondence to: Alexander Kastius <alexander.kastius@hpi.de>.

Accepted at Methods and Opportunities at Small Scale (MOSS), ICML 2025, Vancouver, Canada.

2.1. B-Tree and its Optimizations

B-trees are a data structure long known in database and algorithm literature (Sedgewick & Wayne, 2011). They are used to index large, unsorted data structures. A B-tree offers a higher width per node than binary search trees. This allows the node to directly point to several children, which allows a tree format with a low height and a node block size that can be optimized for read/write performance on lower-tier storage devices. There are several variants that further improve B-trees, namely B* trees and B+ trees (Comer, 1979). They do not fundamentally change the mechanisms for updates. A long track of research exists concerning optimization of the storage of a tree within different storage tiers, like optimizing cache hits when reading large trees from storage, for example, in Rao & Ross (2000). Others aim at optimizing them for nonvolatile memory (Chen & Jin, 2015). One study aims to optimize the internal process of insertion or deletion, as described in Jannink (1995).

2.2. Tree Neural Networks

Parsing sequential inputs is a well-discussed challenge in machine learning research. Up to the rise of transformers, architectures like Recurrent Neural Networks (RNNs) have been used for that purpose, their earliest descriptions are provided in Rumelhart et al. (1986). Further architectural developments led to LSTMs, a seemingly superior architecture for parsing sequences (Hochreiter & Schmidhuber, 1997). Most of those architectures have been superseded by the transformer, see, Vaswani et al. (2017), which incorporates features reused in our model. Ren et al. (2021) displays a project, in which a tree was parsed top-down, with the child nodes receiving the parent nodes' output if the respective node was chosen by a classifier to be the correct element. Cheng et al. (2018) use a comparable architecture that parses each node of the tree by using the values of the left sibling as well as the outermost right child of the current node as input to each iteration step. The dependencies do not allow caching, which is contrary to our setup.

3. Process Model

The following section introduces the general setup of the Markov Decision Process (MDP) that the agent is facing. We will follow the three main components of an MDP: The state $s_t \in S$, the action $a_t \in A$, and the reward r_t. During training, the process is initialized by starting with a randomly generated valid B-tree filled with data and a set of items to append or delete from that tree. The agent then faces the task of determining an order of insertion by selecting an item to either append or delete from that list. After each selection, the operation is carried out without further interaction with the agent.

The state needs to incorporate two pieces of information: The whole tree under assessment, parsed through our hierarchical network, and the list of elements to insert and delete in an array of fixed length. The action choice consists of selecting which of the above-mentioned inserts and deletes has to be executed next. The action space is designed analogously to the list of operations, with one action corresponding to one element in the input vector. Here, Action masking (Huang & Ontañón, 2022) is used to dynamically modify the action space to remove all actions that resemble empty spots in the input.

The reward is based on the execution costs of each operation performed:

$$r_t = -p \cdot (n_{\text{leafsplit}} + n_{\text{leafmerge}}) - q \cdot (n_{\text{split}} - n_{\text{merge}}). \tag{1}$$

The cost of any operation is determined by the number of nodes split n_{split} and $n_{\text{leafsplit}}$ as well as merges n_{merge}, $n_{\text{leafmerge}}$ required to successfully execute that operation. The minimum value of a state thus becomes the minimal discounted number of splits and merges required to insert and delete all values given for that specific state.

4. Representation Model Setup

The decision is performed by an agent whose policy is optimized using Proximal Policy Optimization (PPO) (Schulman et al., 2017). PPO relies on the policy gradient theorem, expanded by clipping the policy gradients. The agent relies on two internal models to implement this algorithm: The value and the policy network. In our use case, the value network should represent the minimal discounted cost of all open insertions and deletions for the given tree. The first layers of both networks are shared and contain our encoder setup. The network is optimized using ADAM (Kingma & Ba, 2015), an overview of the most important hyperparameters is given in Table 3.

Hierarchical Model Architecture A noticeable challenge of this and other related problems is induced by its input. Our solution aims to efficiently embed the whole B-tree at inference time in a live application. Flattening the whole tree and passing it as input to a large network is often intractable due to the increasing number of neurons required in the first layer.

We choose to tackle this challenge by embedding the B-tree recursively, to keep the number of trainable parameters constant while increasing tree size. A full visual description of our hierarchical architecture is displayed in Figure 3, see Appendix B. The root embedding of a B-tree can be calculated as follows:

- We embed leaf values using a linear projection layer.
- For the internal nodes, we concatenate all child embeddings and the node keys.
- To improve performance, we add a positional embedding based on the depth of the node currently being parsed. The positional encoding is a unique learnable parameter for every layer of the tree.
- The aggregated child embeddings are then passed to an attention layer, which computes the embedding of the node currently under assessment. The layer follows the classical layout described by Vaswani et al. (2017):

$$f(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{2}$$

  The inner setup of the module consists of one layer of self-attention, followed by two linear layers as part of the feedforward network. A setup replacing the attention layer with a linear layer was also tested.

Our hierarchical encoder allows us to process nodes within one tree layer in parallel, as each node is only dependent on its child embeddings and its own keys. The final root embedding is then passed to the value and policy network of the agent, which are composed of two linear layers.

Caching The proposed structure allows further optimization. Many operations change only small or even single elements of the tree. The embeddings of all unchanged subtrees below or parallel to the changed node stay the same and thus do not need to be recomputed during operation. If a node is changed throughout the execution of the operation, it is invalidated and its embedding will be recomputed through the next cycle of the forward pass. As displayed in Figure 4, see Appendix C, all nodes along its path to the root need to be invalidated as well. During training, caching is not used as backpropagation needs to propagate the whole tree.

5. Evaluation

We compare our heuristic to the baseline in terms of the highest achieved reward. This does not fully represent caching effects when the tree is stored in permanent storage, but it serves as an indicator of real-world performance. We show that using a tool like this can significantly reduce the number of operations executed throughout a process. Furthermore, several setups are compared, while several baselines are discussed. The training performance observed throughout the experiment provides insights into the requirements of the system during training.

Baselines The baseline is formed by randomly ordering the operations for a given set of inserts and deletes. This resembles executing them in order if received in random order on a real-world system. Furthermore, we evaluated three non-optimal heuristics. The first one, named alternating in the following text, chooses the largest available items in an alternating fashion. Thus, the largest element from the list of available inserts is chosen, and then the largest element is deleted. This is repeated until the list of available deletes and inserts is empty. The second evaluated heuristic, called insert first, first executes all inserts, followed by all deletes in descending order. The third heuristic, called delete first, executes all deletes in ascending order, followed by all inserts. The average reward of random execution and all heuristics is displayed in Table 1.

Heuristic	Avg. Reward (over 10 000 episodes)
Alternating	**-4.28**
Inserts First	-6.44
Deletes First	-8.70
Random	-4.76

Table 1. The average reward of all four heuristics. Interestingly, deleting the elements first causes more reconstructions due to underfilled nodes than randomly inserting. The same goes for executing all inserts first. Random performs worse than alternating.

Evaluation Setup The experiment setup is chosen so that the resulting problems can still be analyzed by a human operator. This setup allows a case-by-case evaluation of our policy in comparison to the heuristics. To achieve this goal, we choose node size and count to be relatively small, as outlined in Table 2. In practice, larger node sizes would be used to maximize write and read performance. Small node sizes also increase the probability of reordering and thus decrease the sparsity of the feedback signal.

For our evaluation we compare the performance of a flattened observation space with a linear feature extractor (Flattened w/ MLP) and an attention-based encoder (Flattened w/ Att.) with our two hierarchical feature extractor variants: Multi-head attention layer after the child combination (HE w/ Att.) and linear layer after child combination (HE w/ MLP). Despite the undesirable scaling properties of the flattened feature extractors, we still wanted to compare architectures on small trees to find out whether our hierarchical architecture can still compete on trees, where all values can be directly parsed into the feature extractor.

Each run lasted for 10^8 episodes. Each episode was executed starting with a randomly generated balanced tree with 24 values, as well as six randomly generated inserts and six randomly generated deletes. The average reward throughout the training process is displayed in Figure 5, see Appendix E. As displayed, our attention-based setup outperforms all other architectures. The average reward of the random policy is -4.76, while the final learning state of the hierarchical feature extractor achieves -2.17, leading to an average improvement of 55% in comparison to randomly selecting elements.

In general, the results displayed by the performance plot also hold true to the distribution of the policies. The hierarchical attention-based setup outperforms all other setups we evaluated for this specific use case. Flattening and using the tree as input yields comparable results, though consistently lower. As discussed earlier, flattening large trees is not viable and only works in small setups, while the best-performing solutions are, in theory, not limited in that regard. A more detailed quantitative and qualitative analysis of the measurements is provided in Appendix E.

Storage Space Recall that we assume that in an applied system, each node takes the full storage space required for the node. (Yao, 1978) has shown that if a tree is filled randomly, in theory, 31% of the keys and pointers stay unused. This leads to the question of whether our approach, by minimizing splits, subsequently also increases the storage efficiency. The experiment setup consisted of a randomly generated tree, which subsequently was filled with six additional elements, while six elements were eventually removed. This process was repeated ten times. We compare the average number of nodes required in this setup with the results achieved by the random heuristic. Figure 9 displays that by using our method, the average storage demand can be temporarily improved by up to 1.06 nodes, or 11%, at the point of maximum difference. This makes this optimization interesting even in those scenarios in which the write/read performance is negligible for the user. While the agent deconstructs this advantage throughout each process, it achieves a noticeably lower result at the end of each 12-step episode.

Inference Time For this system to be practically applicable, it is necessary that the inference is reasonably fast. Table 4 compares the time in milliseconds required for one forward pass on the hardware used to perform the experiments. It shows that a single forward pass, without caching, can be performed in a few milliseconds. In this scenario, all data read is already in the main memory, it is thus assumed that the embeddings are readily available. In this system, a forward pass is fast enough to be worth the computation time when 55% of the otherwise induced write operations can be omitted. For details, see Appendix I.

6. Conclusion

We discovered a model setup that can successfully learn well-performing policies in the provided environment. It is capable of outperforming the execution cost of random execution, as well as any rule-based approach we came up with. We have shown that even without knowledge of the tree, the agent learns a policy that performs well without any necessity to parse the input tree. To achieve optimal results, our agent must effectively parse the input tree. Our hierarchical encoder structure accomplishes this by parsing from the leaves up to the root. We demonstrated that this novel hierarchical approach outperforms all tested alternatives while remaining agnostic to the tree depth and node structure. Additionally, our evaluations highlight its potential for storage optimization, requiring fewer node splits and merges.

As our model is agnostic to the problem it is applied to, exploring other applications in which the input data can be reshaped would display future application possibilities. This would apply to all use cases in which the data can be augmented by ordering it in a semantic hierarchy. Future work should observe if our measurements persist even on very large datasets.

References

Chen, S. and Jin, Q. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, 2015. doi: 10.14778/2752939.2752947. URL http://www.vldb.org/pvldb/vol8/p786-chen.pdf.

Cheng, Z., Yuan, C., Li, J., and Yang, H. Treenet: Learning sentence representations with unconstrained tree structure. In Lang, J. (ed.), *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pp. 4005–4011. ijcai.org, 2018. doi: 10.24963/IJCAI.2018/557. URL https://doi.org/10.24963/ijcai.2018/557.

Comer, D. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979. doi: 10.1145/356770.356776. URL https://doi.org/10.1145/356770.356776.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997. doi: 10.1162/NECO.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Huang, S. and Ontañón, S. A closer look at invalid action masking in policy gradient algorithms. In Barták, R., Keshtkar, F., and Franklin, M. (eds.), *Proceedings of the Thirty-Fifth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2022, Hutchinson Island, Jensen Beach, Florida, USA, May 15-18, 2022*, 2022. doi: 10.32473/FLAIRS.V35I.130584. URL https://doi.org/10.32473/flairs.v35i.130584.

Jannink, J. Implementing deletion in b+-trees. *SIGMOD Rec.*, 24(1):33–38, 1995. doi: 10.1145/202660.202666. URL https://doi.org/10.1145/202660.202666.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Rao, J. and Ross, K. A. Making b^+-trees cache conscious in main memory. In Chen, W., Naughton, J. F., and Bernstein, P. A. (eds.), *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pp. 475–486. ACM, 2000. doi: 10.1145/342009.335449. URL https://doi.org/10.1145/342009.335449.

Ren, X., Gu, H., and Wei, W. Tree-rnn: Tree structural recurrent neural network for network traffic classification. *Expert Syst. Appl.*, 167:114363, 2021. doi: 10.1016/J.ESWA.2020.114363. URL https://doi.org/10.1016/j.eswa.2020.114363.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986. *Biometrika*, 71(599-607):6, 1986.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Sedgewick, R. and Wayne, K. *Algorithms, 4th Edition*. Addison-Wesley, 2011. ISBN 978-0-321-57351-3.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Yao, A. C. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978. doi: 10.1007/BF00289075. URL https://doi.org/10.1007/BF00289075.

A. Tree Balancing Operations

The following two figures display the required rebalancing operations in a B-tree, which is why optimization to minimize the number of operations is necessary.

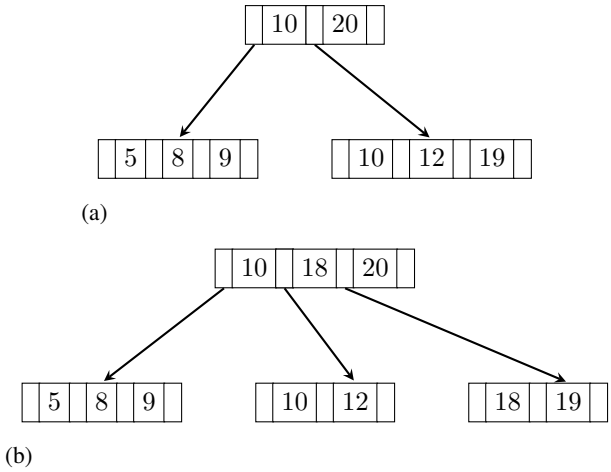

Figure 1. The resulting tree when an already full node on a $b = 3$ tree would require an additional child at key $n = 18$. The outermost node is split, and at least three nodes need to be partly rewritten. This propagates to the parent and could cause additional reorderings if the parent node exceeds its maximum size after executing this operation.

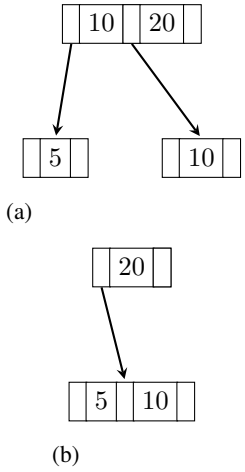

Figure 2. The resulting tree when two neighboring nodes are merged due to both being underfilled, in this case having less than two keys. Again, the resulting write operation causes at least two nodes to be rewritten in storage.

B. Model Architecture

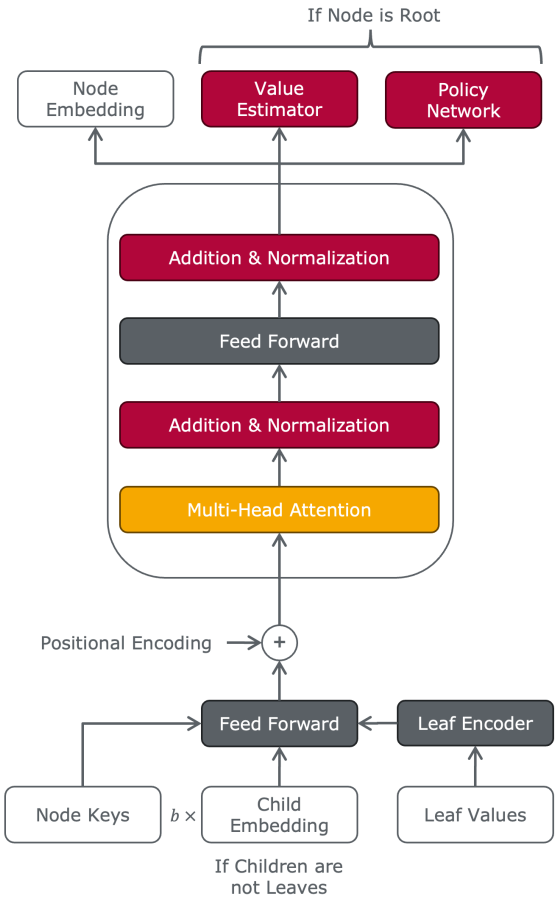

Figure 3. A visualization of our hierarchical embedding setup. A node embedding can be calculated using only its keys and its b children. The resulting values are either propagated upward or passed to the value and policy network once the root of the tree is reached. The output of each node is cached and made available to parents from the cache if it is available and unchanged.

C. Caching

Figure 4 illustrates the caching mechanism, which can be implemented from the leaf to the root node of the tree during inference to avoid recomputation on large datasets.

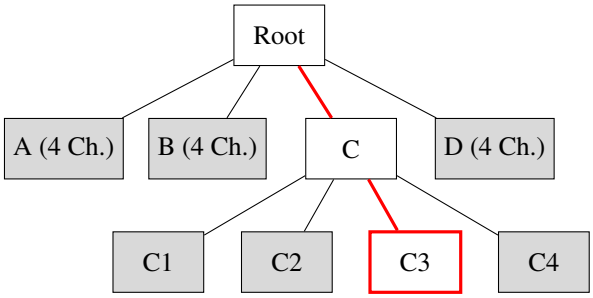

Figure 4. An illustration of the caching mechanism: A change in C3 invalidates all nodes along its path to the root. All gray nodes are read from the cache and do not need to be recomputed. A, B, and D contain four children each. This saves parsing 18 nodes, while only six embeddings need to be loaded from cache or recomputed.

D. Experiment Setup

The following table contains the parameters for the random experiment described in the text.

Parameter	Value
Node-Size b	4
Elements in Tree	24
Number of Insert Ops at s_0	6
Number of Delete Ops at s_0	6

Table 2. Environment variables for experiments.

E. Evaluation Results

A more detailed qualitative analysis was performed to illustrate the space of reachable policies and their average rewards. Figure 6 displays the distribution of quantiles achieved by the agent's chosen solution. The quantiles were computed by estimating the cost of 1,000 permutations of the operations. This process was repeated for 1,000 initial trees. It shows that the system finds a well-performing policy in most cases. An illustration of one of those scenarios is given in Figure 7. A few tree setups still lead to poor resulting performance, as is the case in the example discussed in Figure 8.

Additionally, we validated the ability of the agent to generalize to unknown tree sizes. While increasing the number of values in our tree to 124, the policy trained on trees with size 24, still outperforms the random and the alternating baseline. After increasing the number of values to 1 024, we could not detect a relevant performance improvement over the alternating baseline. The decrease in relative performance might be due to the positional embedding, which encounters unknown tree depths during inference time. Interestingly, this can still be compensated for when only one additional tree level is added while increasing the tree size from 24 to 124. If this trend holds for larger trees, production systems would only need to be fine-tuned after at least a B-fold increase in values.

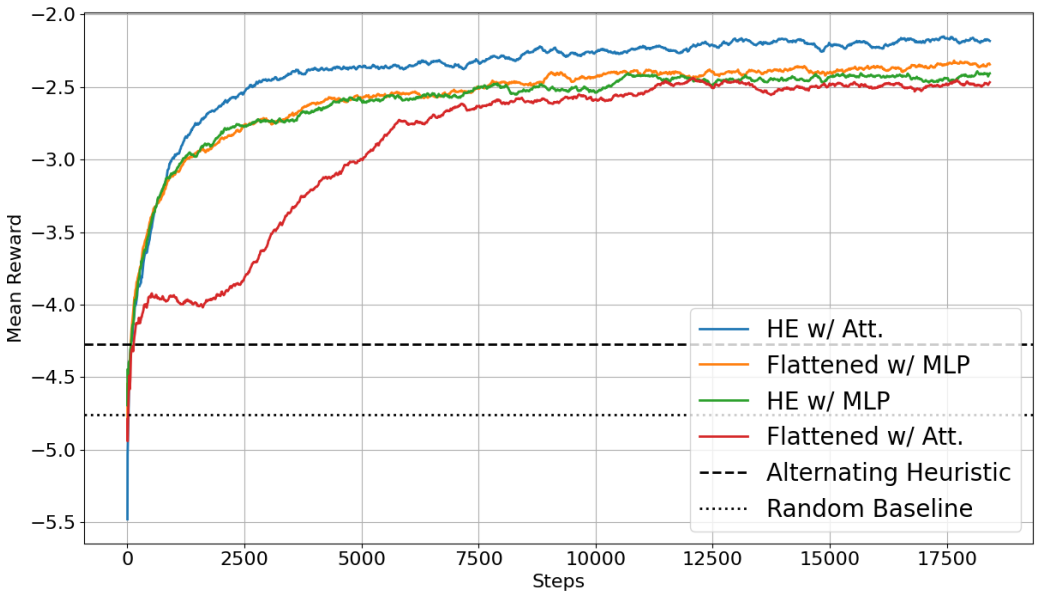

Figure 5. Comparison of the average reward throughout the training runs with our hierarchical attention setup (HE w/ Att.), a hierarchical feature extractor using a feedforward network (HE w/ MLP), and a linear feature extractor that parses the flattened B-tree (Flattened w/ MLP). The fourth setup consists of a model with one attention layer, which receives the flattened tree as input (Flattened w/ Att.). The average rewards of all the heuristics described are shown as horizontal lines.

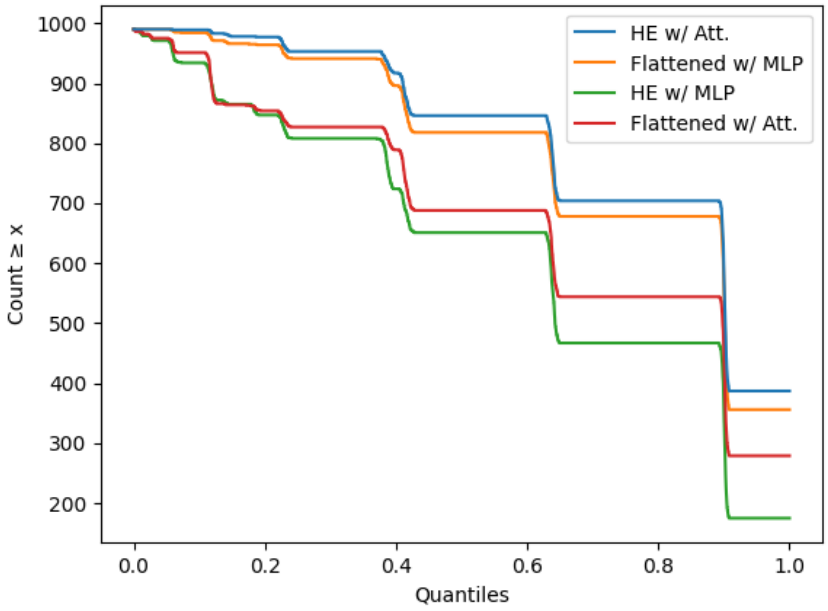

Figure 6. The quantile position of the agent's policy compared to the overall space of possible permutations computed by exhaustive search over 1 000 randomly generated trees. All possible execution orders were aggregated and their reward measured. The best-performing setup achieves a position in the 90% quantile in over 700 of the 1 000 experiments. The steps in the graph are caused by the stepwise reward function of the environment, in which many permutations of operations end up with the same number of overall operations and thus reward. By only improving those by a single reorder a noticeable number of alternative execution orders is outperformed.

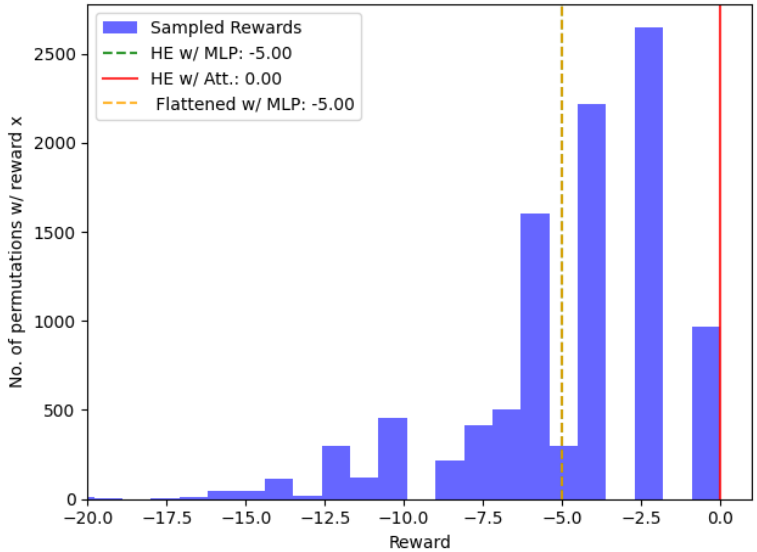

Figure 7. The summed-up received the reward of our best-performing model, red, as well as the flattened input alternative. In comparison, we plot the resulting reward for 1 000 permutations of the 12 executed actions. In this case (seed: 98476), the agent performed exceptionally well when compared to the distribution of possible rewards. A detailed analysis of the input showed no reliable indicator of why this was caused. This quantile position was the foundation of Figure 6. A comparison of this setup and the following one is provided in the caption of Figure 8.

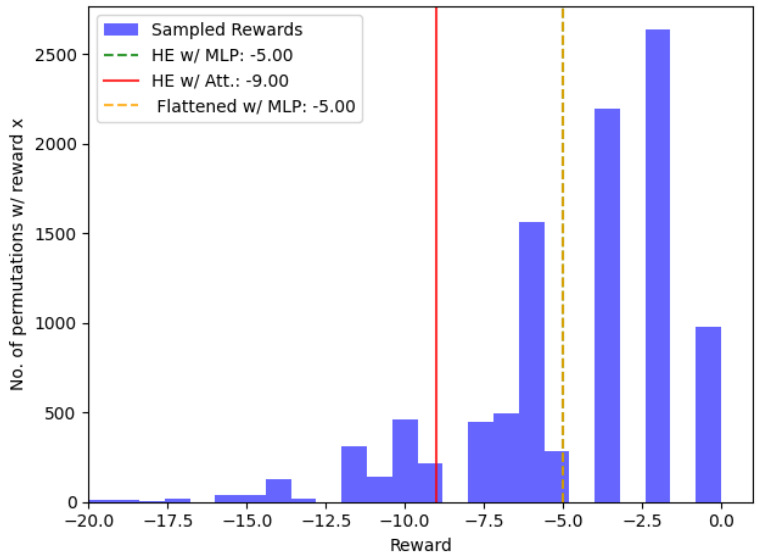

Figure 8. The summed-up received the reward of our best-performing model, red, as well as the flattened input alternative. In comparison, we plot the resulting reward for 1 000 permutations of the 12 executed actions. In this case (seed: 50010), the agent performed exceptionally poorly when compared to the distribution of possible rewards. It seems that the tree structure is at fault, even though there are only marginal differences compared to the tree that formed the initial state in the episode in Figure 7. In this case, the insertion of one value caused two splits and the addition of a full layer to the tree. A costly operation, which could have been avoided but was not properly predicted by the agent. This shows that even small inaccuracies can cause costly operations.

F. Storage Improvements

Figure 9 displays the required storage space for a tree when repeatedly executing 12 evenly balanced insert and delete operations. The agent learns to empty the tree before refilling it and saves overall storage space.

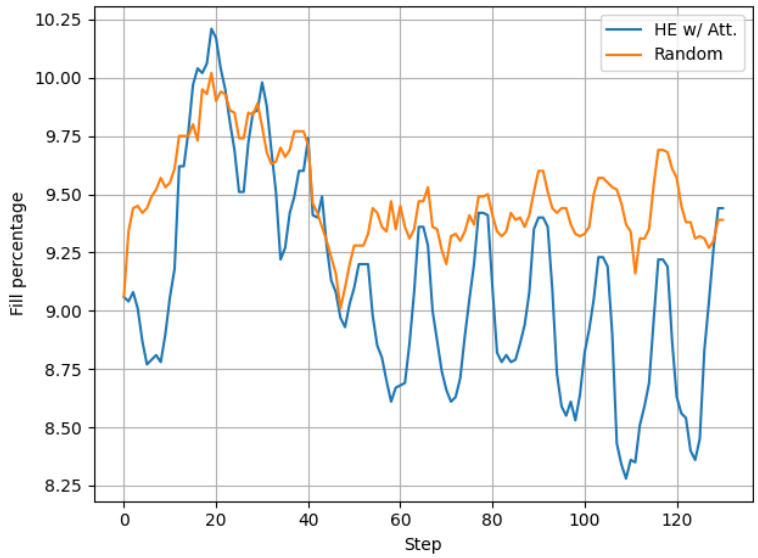

Figure 9. The average number of nodes in the graph according to the experiment described in Section 5.

G. Observed policies

A more detailed qualitative analysis was performed to illustrate the space of reachable policies and their average rewards. Figure 6, see Appendix E, displays the distribution of quantiles achieved by the agent's chosen solution. The quantiles were computed by estimating the cost of 1,000 permutations of the operations. This process was repeated for 1,000 initial trees. It shows that the system finds a well-performing policy in most of the cases, an illustration of one of those scenarios is given in Figure 7, see Appendix E. A few tree setups still lead to poor resulting performance, as is the case in the example discussed in Figure 8. The following figures display the execution order chosen by the agent, which only observes the actions in 40 episodes with 12 steps each. It shows no obvious policy we could reproduce with a rule-based approach. It reliably outperforms each of the heuristics designed and evaluated by us.

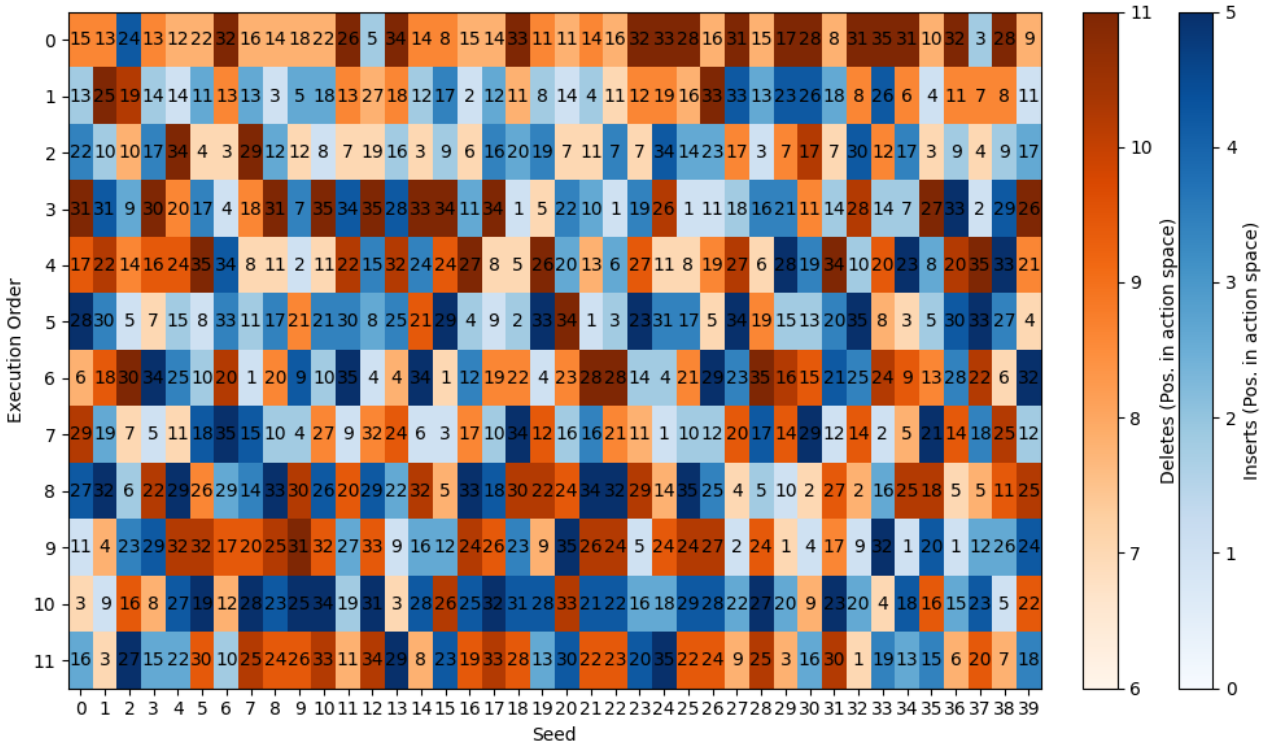

Figure 10. Each column shows the actions for one episode. The color represents whether it is an insert (blue) or a delete (red). This resembles the position of the value in the action space, not in the space of possible values with which each action can be associated. The hue shows the size of the value relative to the others, also encoded in the position in the sorted action. The agent tends to perform inserts and deletes in an alternating fashion, but also stacks multiple operations of the same type from time to time. The inserted values do not always resemble the deleted values, and values in close proximity do not always follow each other. Both patterns can be observed, though. For example, in seed number 29, 15 gets inserted, then 16 and 14 get deleted, then 10 gets inserted.

H. Hyperparameters

The following table includes the hyperparameters used to conduct the experiments.

MDP	
Gamma	0.999
Optimizer	
Step-Size	0.1
Architecture	
Embedding Size	64
Num. Feedforward Layers	2
Num. Attention Heads	4
Activation Function	ReLu
Batch Size	512

Table 3. Excerpt of parameter choices for the learning setup as well as the optimizer.

I. Inference Time

The following table displays the inference time for various tree sizes. The experiments were performed on a system using AMD CPUs and an Nvidia A40. It displays that the inference time is small enough, so that in many cases the performance gain outweighs the cost of estimation.

# Values per Tree	Runtime (ms)
24	1.89
124	2.42
1024	7.21

Table 4. The average execution time for a forward pass in all attention-based setups. It grows in logarithmic time.