NOT PUBLISHED

Beyond Balanced: Learning Static Search Trees with Tree-MDPs

Konrad Gerlach [*1] **Till Zemann** [*1] **Constantin Kühne** [*1] **Hendrik Droste** [*1] **Alexander Kastius** [*1]
Rainer Schlosser [1] **Ralf Herbrich** [1]

Abstract

Search algorithms are a fundamental component of any sophisticated data-handling system. We propose training a model to find optimal lookup points, leading to pre-generated static binary trees. These trees are constructed by framing the problem as a Tree-Markov Decision Process, enabling adaptation to a given data query distribution. The generated trees then lead to optimized lookup times given a defined request and response pattern.

1. Introduction

Database systems and data structures heavily rely on efficient search. Solution algorithms like binary search are long discussed and regularly used (Sedgewick & Wayne, 2011). They maintain height balance to guarantee a worst-case of $O(\log \mathfrak{n})$ lookups regardless of the query distribution. However, real-world applications — for example shopping websites with items of different popularities — often have highly skewed access patterns (Zhang & Ross, 2022). We propose to develop an adaptive system that is able to generate the representation of a search algorithm given the distribution of data queries. Our method automatically learns to place more frequently accessed data higher in the tree. In our preliminary experiments, the learned trees improve average search costs compared to binary search on Gaussian and Exponential distributions by up to 19.34% and 40.26% respectively (see Table 2). These results can be reproduced using our accompanying notebook. Our work is highly related to search trees, which we use as a representation, as well as indices, which serve a comparable task for databases. The system proposed is small enough to run in parallel to the database management system and can be trained ad-hoc in live environments without requiring additional specialized hardware. The search tree representing binary search always splits the data in half. If the data or query distribution is very skewed, starting at a different point could yield faster lookups. This holds for deeper levels of the tree as well. We aim to pregenerate a binary search tree (BST) that is optimal with regard to the distribution of data as well as the distribution of the queries.

2. Related Work

The construction of efficient BSTs is a well-studied problem. The foundational work in constructing exact static optimal BSTs was presented by Knuth (Knuth, 1971). He defined the problem formally, considering probabilities for successful searches (searched element is contained in the tree) and unsuccessful searches (searched element is not contained in the tree). Knuth discusses a dynamic programming (DP) algorithm in $O(\mathfrak{n}^3)$ and an improved version in $O(\mathfrak{n}^2)$ time complexity and $O(\mathfrak{n}^2)$ space complexity that guarantees finding the tree with the absolute minimum weighted path length given these probabilities. However, for a large number of keys, which is usual in database settings, this exact optimal algorithm can still be impractical. Further research thus focused on faster approximation methods. Knuth proposed heuristics, including one that greedily places the most frequent key as the split key and another that chooses the split key which splits the key distribution in half such that the left and right subtrees have the same probability mass (Knuth, 1971). Mehlhorn (Mehlhorn, 1971) analyzed these heuristics, showing that the weight-balancing approach (Rule II) consistently produces nearly optimal trees. This heuristic can be implemented efficiently in $O(\mathfrak{n} \log \mathfrak{n})$ or even $O(\mathfrak{n})$ time (Mehlhorn, 1971), which makes it a practical near-optimal solution.

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

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

A different approach to generating static search trees are *dynamic* self-adjusting trees, such as Splay Trees (Sleator & Tarjan, 1985). They do not address the static BST generation problem and can thus not be referenced as a baseline. Dinitz et al. (2024) displays that learning the data distribution can be of use for binary search, but without fully generating the BST. A comparable concept was presented for indices in (Kraska et al., 2018). The tree-MDP framework introduced by Scavuzzo et al. (2022) provides a generalization of temporal MDPs that is well-suited for tree-structured decision making. They show that tree-MDPs can make RL effective for combinatorial problems. We adapted this formulation to generate BSTs.

3. Search Tree Optimization

We formally define the problem as follows. As input we have a sorted list of $\mathfrak{n}$ distinct index elements $E = [E_0, E_1, \ldots, E_{\mathfrak{n}-1}]$. They correspond to keys K_{E_i} in a base array K such that for all $0 \leq i < \mathfrak{n}$, we have $E_i < E_{i+1}$ and $K_{E_i} < K_{E_{i+1}}$). We also have a query count vector $N = (N_0, N_1, \ldots, N_{\mathfrak{n}-1})$, where N_i is the number of times the key corresponding to index E_i was queried, which is obtained from a query log Q. After running our RL pipeline, the output is an optimized static binary search tree $\mathcal{T}$, where the nodes contain the indices from E, which can be interpreted as pointers to the corresponding data values, e.g., on disk. Our objective is to minimize the total search cost $c(\mathcal{T})$, see Equation 1, associated with this tree under the given query counts N.

Binary search trees as splitting policies. We generate a BST, which explicitly defines a static splitting policy or, in other words, a search algorithm. A binary search tree $\mathcal{T}$ over a sorted list of $\mathfrak{n}$ distinct indices $E = [E_0, E_1, \ldots, E_{\mathfrak{n}-1}]$ is a rooted binary tree where each node contains an index E_i. We denote the index contained in the root node of a tree (or subtree) $\mathcal{T}$ as element $e(\mathcal{T})$. In a binary search tree, it always holds that for any node containing index $E_i = e(\mathcal{T})$, all indices in the left subtree $l(\mathcal{T})$ are less than E_i, and all indices in the right subtree $r(\mathcal{T})$ are greater than E_i. A distinct advantage of our method is that, after tree construction, we no longer need to maintain the sorted index array or key array in memory. Instead, we can use a bijective mapping from tree nodes to their corresponding key values stored elsewhere (e.g., on disk). This indirection allows us to access the keys via pointers when needed, thus saving memory.

A binary search tree fully specifies a static search strategy. When searching for a queried key q, the search starts at the root. At each node containing index $E_i = e(\mathcal{T})$, the algorithm compares q with the corresponding key K_{E_i}. If $q = K_{E_i}$, the search terminates successfully. If $q < K_{E_i}$, the search continues in the left subtree $l(\mathcal{T})$; if $q > K_{E_i}$, it continues in the right subtree $r(\mathcal{T})$. Therefore, the choice of the index E_i for the root of a (sub)tree $\mathcal{T}$ acts as the splitting rule for the set of indices contained in $\mathcal{T}$. Our RL approach aims to optimize these splitting rules for the given query counts.

Query Cost: We define the cost of a search query for a specific index E_i in a binary search tree $\mathcal{T}$ as the number of edges traversed from the root to reach the node containing E_i. This is equivalent to the depth of the node containing E_i, denoted as $d_{\mathcal{T}}(E_i)$. The root node has a depth of zero ($E_i = e(\mathcal{T})$). Thus, our cost represents the number of edge traversals required to find the index in the tree structure.

Total Query Cost: The total cost of a binary search tree $\mathcal{T}$ is defined as the sum of the query costs for all queries recorded in the query log, weighted by their counts:

$$c(\mathcal{T}) = \sum_{i=0}^{\mathfrak{n}-1} N_i \cdot d_{\mathcal{T}}(E_i) \tag{1}$$

This is a slightly perturbed variant of the cost function in Knuth (1971). We do not consider misses, use absolute instead of relative frequencies and define the root node as level zero instead of one. This total cost $c(\mathcal{T})$ represents the sum of edges traversed over all searches represented by the query counts N. Minimizing this value means minimizing the total work done by the search index over the historical queries, which can be linked to minimizing the total number of disk reads or other costly operations, thus improving the overall search time.

The objective of finding the optimal BST $\mathcal{T}^*$ can be written in terms of the expected number of comparisons per query as $\mathcal{T}^* = \arg\min_{\mathcal{T}} \sum_{i=0}^{\mathfrak{n}-1} N_i \cdot d_{\mathcal{T}}(E_i) = \arg\min_{\mathcal{T}} c(\mathcal{T})$. See Appendix A for further information.

Subtree Cost Calculation: Instead of calculating the total cost $c(\mathcal{T})$ described in Equation 1 in a slow brute-force algorithm, we can recursively compute it using a bottom-up approach based on the query counts $N = (N_0, N_1, \ldots, N_{\mathfrak{n}-1})$, very similar to Knuth (1971). To compute $c(\mathcal{T})$ recursively, we calculate two values for any subtree $\mathcal{T}'$ within the tree $\mathcal{T}$.

The two recursively calculated values are: $n(\mathcal{T}')$, the count of queries searching for indices within that subtree, and $c(\mathcal{T}')$, the contribution of the subtree to the total cost. Their computation is described in detail in Appendix B.

Using these definitions, the total cost for the entire tree $\mathcal{T}$ can be calculated by recursively computing $c(\mathcal{T}')$ and $n(\mathcal{T}')$ starting from the leaves up to the root, see Figure 3 in Appendix C. The final result for the whole tree is simply the value $c(\mathcal{T})$ computed for the root node, which finally computes Equation 1.

4. Process Model

We propose to use a tree-MDP identical to the formulation by Scavuzzo et al. (2022) for the BST generation problem. In this formulation, an action $a = k$ taken in state $s = (low, high)$ does not transition to a single next state s_{t+1}, but rather branches into two independent subsequent subproblems, $s_l = (low, k-1)$ and $s_r = (k+1, high)$, corresponding to the left and right child subtrees. The value of state s is defined recursively based on the values of these independent subproblems. The discount factor γ is applied at each level of this recursion. Specifically, a reward generated d levels below the node corresponding to state s is discounted by γ^d.

For this tree-MDP, the objective is to find a policy π^* that maximizes the expected value $V^*(s_0)$ from the initial state $s_0 = (0, \mathfrak{n}-1)$, where the optimal value function $V^*(s)$ satisfies the recursive Bellman-like Equation 10. When $\gamma = 1$, optimizing $V^*(s_0)$ is equivalent to minimizing $c(\mathcal{T})$. When $\gamma < 1$, deeper trees are slightly favored because of the exponential down-weighting of cost contributions from deeper nodes, even if the absolute total cost $c(\mathcal{T})$ is slightly higher. In our experiments, we use $\gamma = 1$ to directly optimize $c(\mathcal{T})$. The MDP is fully specified through state, action, and reward:

- **State Space $\mathcal{S}$:** A state $s \in \mathcal{S}$ represents a subproblem defined by a half-open interval of indices $s = [low, high)$ within the sorted list E, corresponding to the sub-array $E[low \dots high-1]$. The initial state is $s_0 = [0, \mathfrak{n})$. A state is terminal when $low \geq high$ and a leaf is reached.

- **Action Space $\mathcal{A}(s)$:** In a non-terminal state $s = [low, high)$ where $low < high$, an action $a \in \mathcal{A}(s)$ consists of choosing a split index k such that $low \leq k < high$. This selects E_k as the root of the subtree for the interval $[low, high)$ and defines two subsequent subproblems: the left subproblem $[low, k)$ and the right subproblem $[k+1, high)$.

- **Reward $\mathcal{R}(s_t, a_t)$:** The reward r_t for action (split) $a_t = k$ in state $s_t = [low, high)$ is the negative cost of pushing queries one level deeper:

$$\mathcal{R}(s_t = [low, high), a_t = k) = -\left(n(l(\mathcal{T}')) + n(r(\mathcal{T}'))\right) \quad = -\left(\sum_{i=low}^{k-1} N_i + \sum_{i=k+1}^{high-1} N_i\right), \tag{2}$$

  where $\mathcal{T}'$ is the subtree rooted at E_k covering the interval $[low, high)$ and $n(\cdot)$ is the sum of query counts in the subtrees, see Equation 7.

Using tree-MDP Scavuzzo et al. (2022), makes it possible to apply $\gamma_{\text{tree}} \leq 1$ with depth-based discounting. The state and action spaces, as well as the reward, are defined in the common definitions. For the tree-MDP, an episode begins at $s_0 = [0, \mathfrak{n})$. The states are recursively split using the selecting split points (actions) until all subproblems (branches of the tree) reach terminal states ($low \geq high$). After episode termination, we have generated a complete tree $\mathcal{T}$. All episodes are finite as the intervals get smaller at each step. Furthermore, a tree-MDP $M_{\text{tree}} = (\mathcal{S}, \mathcal{A}, \mathcal{P}_{\text{tree}}, \mathcal{R}, \gamma_{\text{tree}})$ is defined by:

- **Transition $\mathcal{P}_{\text{tree}}(s_{t+1}|s_t, a_t)$:** Equivalently to the sequential MDP, a state $s_t = [low, high)$ deterministically branches out into two subproblems $[low, k)$ and $[k+1, high)$ for action $a_t = k$. The only difference is that a sequence within the tree-MDP only visits future states within the subtree $\mathcal{T}'$ rooted at E_k that covers the interval $s_t = [low, high)$. Also, similarly to the MDP, leaf nodes represent terminal states. Here, a leaf node is the terminal state of one branch of the tree-MDP.

- **Discount Factor γ_{tree}:** Can be chosen as $\gamma_{\text{tree}} \in [0, 1]$. It is applied at each level of the recursion to discount based on the depth relative to the current node. If $\gamma_{\text{tree}} = 1$, this formulation again minimizes the total cost.

We can also specify the exact optimal value function $V^*(s)$ for the tree-MDP, see Appendix D for further information.

5. Optimization Algorithm

We use A2C for optimization, which is the synchronous variant of A3C described in Mnih et al. (2016). We enhanced the algorithm to use entropy regularization, an idea first outlined by Haarnoja et al. (2018). The entropy bonus coefficient β is linearly decayed from $\beta_{start} = 0.3$ down to $\beta_{end} = 0.02$. The network uses a value-estimator, as well as an actor network, which represents the agents' decisions. Adam is used to optimize the network parameters (Kingma & Ba, 2015).

In this setup, the actor outputs logits for a predefined number of discrete actions, $|\mathcal{A}|$. These logits are mapped to probabilities with a Softmax function and define a categorical distribution over action indices $idx \in \{0, \dots, |\mathcal{A}| - 1\}$. We chose $|\mathcal{A}| = 10$ in our experiments. This chosen index idx is then mapped to a split point k within the valid index range for the current state $s = [low, high)$, which is $\{low, \dots, high - 1\}$. To ensure the split point k is always a valid index within this range, we map the discrete action index idx linearly to the continuous interval $[low, high - 1]$ and round to the nearest integer:

$$k = \text{round}\left(low + \frac{idx}{|\mathcal{A}| - 1} \cdot (high - 1 - low)\right). \tag{3}$$

Here, if $idx = 0$, the formula gives $k = \text{round}(low) = low$. If $idx = |\mathcal{A}| - 1$, it gives $k = \text{round}(low + high - 1 - low) = \text{round}(high - 1) = high - 1$. Thus, the generated k always lies in the desired range $[low, high - 1]$. Note that using a fixed number of discrete actions $|\mathcal{A}|$ introduces a tradeoff: a smaller $|\mathcal{A}|$ simplifies the learning problem but limits the policy's ability to select any arbitrary index, potentially preventing it from finding the truly optimal split if it falls between the representable points, especially for large intervals $(high - low)$. A larger $|\mathcal{A}|$ offers finer control but increases the complexity of the output space and thus may take longer to train.

6. Experiments

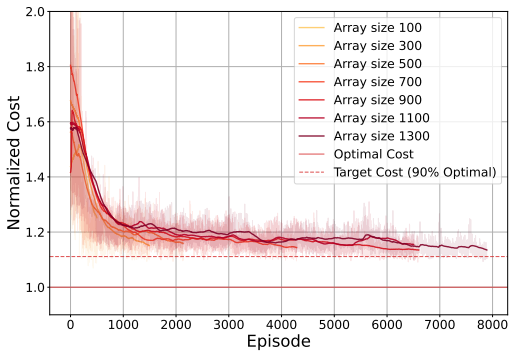

Figure 1. Learning curves for different array sizes with an exponential moving average ($\alpha = 0.005$) overlayed per learning curve for visual clarity. Each run stops when three episodes in a row generate a BST that reaches at least 90% of the optimal reward.

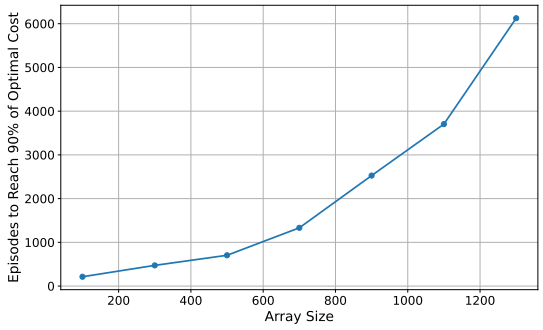

Figure 2. Number of training episodes required to reach 90% of the optimal search tree's reward. We stop when three consecutive training episodes, which are stochastic due to exploration, achieve 90% optimality.

To evaluate the scaling performance of our approach, we trained the discrete A2C agent in the tree-MDP with $\gamma_{\text{tree}} = 1.0$. We used array sizes $[100, 300, \dots, 1300]$ with queries drawn from a Gaussian distribution ($\mu = |E|/2$, $\sigma = |E|/4$). For the learning curves shown in Figure 1, the training was stopped early when the agent achieved a total tree cost within 90% of the optimal cost for three consecutive episodes. We can see in Figure 2 that the number of episodes required to reach 90% of the optimal cost increases with the array size, which is expected. The scaling is worse than linear (at least quadratic), though further scaling experiments with larger arrays are needed to confirm this.

7. Conclusion

We present a reinforcement learning approach for constructing near-optimal static binary search trees intended for database indexing. It constructs the static binary search tree for a given list of unique keys and known query counts. We show that we can efficiently compute a near-optimal search tree for a given load, and use this tree to index the data. We think that building

on top of our RL pipeline is a promising approach for practical applications, especially for skewed distributions. For this, we propose to improve the RL algorithms used in this work and to benchmark them with end-to-end latency measurements for databases, which would achieve valuable insights and might improve the existing heuristic algorithms with a more principled optimization approach.

References

Dinitz, M., Im, S., Lavastida, T., Moseley, B., Niaparast, A., and Vassilvitskii, S. Binary search with distributional predictions. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/a4b293979b8b521e9222d30c40246911-Abstract-Conference.html.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Dy, J. G. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1856–1865. PMLR, 2018. URL http://proceedings.mlr.press/v80/haarnoja18b.html.

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.

Knuth, D. E. Optimum binary search trees. *Acta Inf.*, 1(1):14–25, March 1971. ISSN 0001-5903. doi: 10.1007/BF00264289. URL https://doi.org/10.1007/BF00264289.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In Das, G., Jermaine, C. M., and Bernstein, P. A. (eds.), *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pp. 489–504. ACM, 2018. doi: 10.1145/3183713.3196909. URL https://doi.org/10.1145/3183713.3196909.

Mehlhorn, K. Nearly optimal binary search trees. *Acta Informatica, v.5, 287-295 (1975)*, 5, 01 1971. doi: 10.1007/BF00264563.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning, 2016. URL http://proceedings.mlr.press/v48/mniha16.html.

Scavuzzo, L., Chen, F., Chetelat, D., Gasse, M., Lodi, A., Yorke-Smith, N., and Aardal, K. Learning to branch with tree mdps. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 18514–18526. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/756d74cd58592849c904421e3b2ec7a4-Paper-Conference.pdf.

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

Sleator, D. D. and Tarjan, R. E. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985. ISSN 0004-5411. doi: 10.1145/3828.3835. URL https://doi.org/10.1145/3828.3835.

Zhang, W. and Ross, K. A. Exploiting data skew for improved query performance. *IEEE Trans. Knowl. Data Eng.*, 34(5):2176–2189, 2022. doi: 10.1109/TKDE.2020.3006446. URL https://doi.org/10.1109/TKDE.2020.3006446.

A. Number of Comparisons Versus Query Cost

We want to minimize the cost $c(\mathcal{T}) = \sum_{i=0}^{\mathtt{n}-1} N_i \cdot d_{\mathcal{T}}(E_i)$ with depths $d_{\mathcal{T}}(E_i)$. This objective is equivalent to minimizing the expected number of *comparisons* per search query. This optimization goal was also used by Knuth (1971), who termed it finding trees with *minimum weighted path length*.

Knuth defines the level of the root as one, while our depth $d_{\mathcal{T}}(E_i)$ starts at zero. The number of key comparisons required to locate an element E_i in a tree $\mathcal{T}$ is $d_{\mathcal{T}}(E_i) + 1$. This precisely corresponds to the level of the node in Knuth's terminology. Therefore, if we consider a queried index q drawn according to the query frequencies N (so that $P(q = E_i) = N_i / \sum_j N_j$), the expected number of comparisons per query is:

$$\mathbb{E}_{q\sim N}[d_{\mathcal{T}}(q) + 1] = \sum_{i=0}^{\mathtt{n}-1} \frac{N_i}{\sum_{j=0}^{\mathtt{n}-1} N_j} \cdot (d_{\mathcal{T}}(E_i) + 1) \tag{4}$$

We can rewrite this using our total cost $c(\mathcal{T})$ (defined in Equation 1) and the total number of queries $n(\mathcal{T}) = \sum_{j=0}^{\mathtt{n}-1} N_j$:

$$\begin{aligned} \mathbb{E}_{q\sim N}[d_{\mathcal{T}}(q) + 1] &= \frac{\sum_{i=0}^{\mathtt{n}-1} N_i \cdot (d_{\mathcal{T}}(E_i) + 1)}{\sum_{j=0}^{\mathtt{n}-1} N_j} \\ &= \frac{\sum_{i=0}^{\mathtt{n}-1} N_i d_{\mathcal{T}}(E_i) + \sum_{i=0}^{\mathtt{n}-1} N_i}{\sum_{j=0}^{\mathtt{n}-1} N_j} \\ &= \frac{c(\mathcal{T}) + n(\mathcal{T})}{n(\mathcal{T})} = \frac{c(\mathcal{T})}{n(\mathcal{T})} + 1 \end{aligned} \tag{5}$$

Since $n(\mathcal{T})$ is a constant value determined by the input query counts N, minimizing the expected number of comparisons (see Equation 5) is equivalent to minimizing our primary objective function $c(\mathcal{T})$. Therefore, finding the optimal BST $\mathcal{T}^*$ can be written as minimizing $c(\mathcal{T})$:

$$\mathcal{T}^* = \arg\min_{\mathcal{T}} \sum_{i=0}^{\mathtt{n}-1} N_i \cdot d_{\mathcal{T}}(E_i) = \arg\min_{\mathcal{T}} c(\mathcal{T}) \tag{6}$$

B. Subtree Cost Components

Two components need to be known to determine the cost of execution for a subtree $\mathcal{T}'$:

1. $n(\mathcal{T}')$: The total count of queries for indices contained within the subtree $\mathcal{T}'$. Formally, we can write it as the sum of query counts N_i over all indices E_i contained in the subtree $\mathcal{T}'$:

   $$n(\mathcal{T}') = \sum_{i=0}^{\mathtt{n}-1} N_i \cdot \mathbb{I}\{E_i \in \mathcal{T}'\} \tag{7}$$

   where $\mathbb{I}\{\cdot\}$ is the indicator function. For the recursive computation, let $E_j = e(\mathcal{T}')$ be the index at the root of $\mathcal{T}'$. The base case is when $\mathcal{T}'$ is a leaf node (containing only E_j), then $n(\mathcal{T}') = N_j$. The recursive step is:

   $$n(\mathcal{T}') = n(l(\mathcal{T}')) + n(r(\mathcal{T}')) + N_j \tag{8}$$

   where $l(\mathcal{T}')$ and $r(\mathcal{T}')$ are the left and right child subtrees of $\mathcal{T}'$. If $\mathcal{T}'$ does not have a left or right child (i.e., $l(\mathcal{T}') = \varepsilon$ or $r(\mathcal{T}') = \varepsilon$), the respective query count sum $n(\cdot)$ for that side is set to zero ($n(\varepsilon) = 0$).

2. $c(\mathcal{T}')$: The contribution of the subtree $\mathcal{T}'$ to the total cost, specifically the sum of weighted depths within the subtree relative to its own root. If $\mathcal{T}'$ is a leaf node, our base case is $c(\mathcal{T}') = 0$. If $\mathcal{T}'$ is an internal node (i.e., not a leaf), we apply the recursive step:

   $$c(\mathcal{T}') = c(l(\mathcal{T}')) + c(r(\mathcal{T}')) + n(l(\mathcal{T}')) + n(r(\mathcal{T}')) \tag{9}$$

   with left child subtree $l(\mathcal{T}')$ and right child subtree $r(\mathcal{T}')$. If $\mathcal{T}'$ does not have a left or right child, the respective cost $c(\cdot)$ and query count sum $n(\cdot)$ are set to zero ($c(\varepsilon) = 0, n(\varepsilon) = 0$). The term $n(l(\mathcal{T}')) + n(r(\mathcal{T}'))$ is the added cost

because all queries directed to the left subtree (total count $n(l(\mathcal{T}'))$) and to the right subtree (total count $n(r(\mathcal{T}'))$) must traverse one extra edge (from the root $e(\mathcal{T}')$ of the current subtree $\mathcal{T}'$ to the root of the child subtree). This increments their depth $e(\mathcal{T}')$ by one. The term $c(l(\mathcal{T}')) + c(r(\mathcal{T}'))$ recursively aggregates the costs from the subtrees below.

C. Tree Process Representation

Figure 3 presents an example of the computation outlined in Section 3.

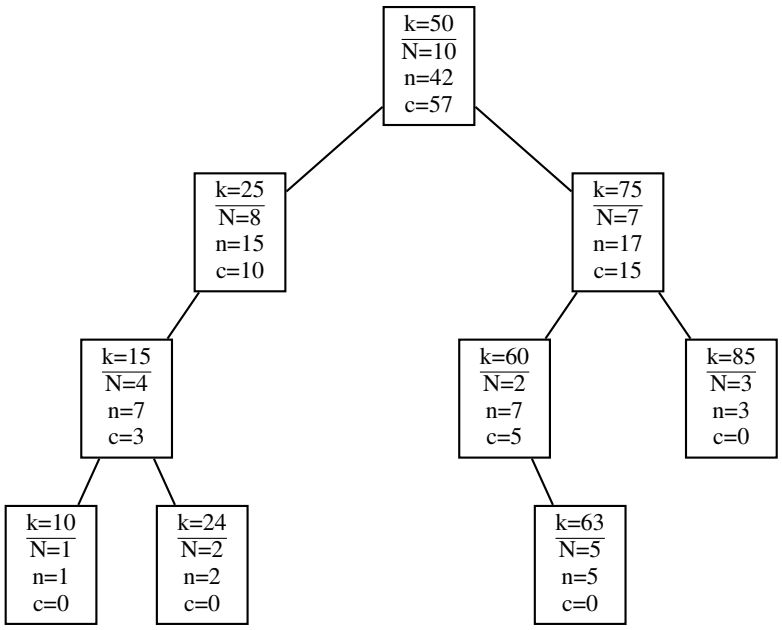

Figure 3. Example binary search tree cost calculation showing node query counts N_i (where i is the index of node v_i that holds key k_i), summed subtree query counts n, and subtree costs c. The values of n and c are calculated using the recursive formula described in Equation 9. For better readability, we leave out subscripts ($i = 0, \ldots, \mathfrak{n} - 1$) for the keys and counts written in the nodes. The cost for the shown example tree $\mathcal{T}$ is $c(\mathcal{T}) = 57$. This cost represents the total number of edges traversed for all queries (see Equation 1). To calculate the example yourself, start from the leaf nodes (where $c = 0$ and $n = N_i$) and apply Equation 8 and Equation 9 from the bottom up until you reach the root node.

D. Value Function Definition

The optimal value of state $s_t = [low, high)$ in the tree-MDP follows a recursive Bellman-like equation. An action $a_t = k$ gives a reward $\mathcal{R}(s_t, a_t)$ and branches the tree into two independent subproblems $[low, k)$ and $[k + 1, high)$. Then, the optimal value function $V^*(s_t)$ is:

$$V^*(s) = \begin{cases} 0 & \text{if } low \geq high \\ \max\limits_{k \in [low, high)} \Big[\mathcal{R}(s,k) + \gamma_{\text{tree}} \big(V^*(s_l(k)) + V^*(s_r(k)) \big) \Big] & \text{if } low < high \end{cases} \tag{10}$$

where states with $low \geq high$ are terminal states (leaf nodes) and states with $low < high$ are non-terminal states.

E. Hyperparameters

Table 1. Hyperparameters used in experiments outperforming binary search by 19.34% and 40.26% on a Gaussian and Exponential distribution, respectively.

Hyperparameter	**Value**
Training Configuration	
Number of training episodes	500
Random seed	42
Plotting rolling window size	50
Tree-MDP	
Array size (number of unique keys)	10
Range of possible queried indices	$[0, 10]$
Discount factor (γ_{tree})	1.0
Advantage Actor-Critic (A2C)	
Hidden layer size	256
Actor learning rate	10^{-4}
Critic learning rate	10^{-5}
Initial entropy coefficient (β_{start})	0.3
Final entropy coefficient (β_{end})	0.02
Optimizer	Adam (Kingma & Ba, 2015)
Adam Hyperparameters	$\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e-8$
Gradient Clipping (L2-Norm)	0.1

F. Experimental Results

Table 2. Improvement of expected number of lookups over binary search for two distribution types. Full experimental details are available in the accompanying notebook.

Distribution	**Training Episodes**	**Array Size**	**Improvement over Binary Search (%)**
Gaussian ($\mu = 0.1\|E\|, \sigma = 0.3\|E\|$)	500	10	19.34
Exponential ($\lambda = 1.5\|E\|$)	500	10	40.26