Transforming Sorted Arrays to Binary Search Trees using Reinforcement Learning

Frederic Sadrieh, Ennio Strohauer, and Helena Ullrich

Hasso-Plattner-Institute, Prof.-Dr.-Helmert-Straße 2-3, 14482 Potsdam
firstname.lastname@student.hpi.uni-potsdam.de

Abstract. This report presents a reinforcement learning approach to constructing binary search trees (BSTs) from sorted arrays, challenging the classical "midpoint" method. We formulate the task as a custom environment in which a PPO agent sequentially programs a computational machine to position values in a BST-encoded array. A specially tailored reward function, using incremental scoring and distance-based penalties, guides the agent to valid BST placements even when partial solutions are encountered. We constrain the search space by restrictive action masking to prohibit invalid or nonsensical moves. Experimental results highlight the importance of reward shaping and action masking in stabilizing training. Our results demonstrate that a highly optimized learning environment is more important than the agent's parameters. Although performance can be improved by using a transformer encoder. Our approach faces challenges when attempting to use more sophisticated control flow and longer input lengths. For readers interested in replicating or extending these experiments, our implementation can be found in our GitHub repository[1].

1 Introduction

Binary Search Trees (BSTs) are an integral part of many applications that require efficient insert, search, and delete operations. Ideally, these operations run in $O(\log n)$ time for a balanced BST, but can degrade to $O(n)$ for skewed or unbalanced trees. A popular way to create a balanced BST is to first sort the array. From the sorted array, one can choose the midpoint as the root, and then recursively place subsequent elements in the left and right subarrays. This approach neatly produces a height-balanced tree in $O(n)$ time, making it a reliable technique in many software libraries and algorithm textbooks.

Despite the existence of this "textbook" algorithm, we investigate whether a reinforcement learning (RL) approach can discover a method that is either comparably efficient or potentially faster under certain constraints or data distributions. In particular, machine learning (ML) methods have proven capable of discovering strategies or heuristics that deviate from traditional algorithmic logic, yet produce competitive or superior results in specialized scenarios [1]. For

[1] https://github.com/EnnioEnnio/rlad_bst

example, when partial information is available at different stages, an adaptive, learned policy could outperform a fixed analytical approach.

Furthermore, if an RL agent can learn to generate valid BSTs from sorted inputs, it suggests a broader feasibility of training agents to generate or optimize other complex algorithms with minimal human supervision. By carefully defining the environment, action space, and reward function, we aim to demonstrate how RL can gradually discover the necessary sequence of operations to place each node in its correct position in the final BST representation.

In the following report, Section 2 details our methodological choices, covering our environment definition, reward function design, agent architecture, and hyperparameter configurations, Section 3 describes our experimental setup and evaluation protocols, Section 4 then presents and analyzes the experimental results, and Sections 5 to 7 offer conclusions, discuss limitations of the project, and outline directions for future work.

2 Methods

In this section, we describe the design choices and technical implementation details used to train a reinforcement learning agent to transform sorted arrays into valid BSTs. We focus on defining the environment, designing the reward function, structuring the architecture of the agent, and selecting appropriate hyperparameters.

2.1 Environment definition

The agent should learn to write code to solve the well-known problem of creating a BST from a sorted list[2]. Since the values of the numbers in the array do not matter, the array will always contain the numbers from 1 to the length of the array. The agent should solve this problem by writing the correct solution into a second array. The second array is interpreted as a BST in array representation, i.e. the root is at index 0 and each node i has its left child at $2i + 1$ and its right child at $2i + 2$. The value 0 is reserved for unwritten places in the second array.

Computing Machine Our environment models a custom "computing machine" that executes a sequence of commands to transform a sorted input array into a BST-encoded output array. The machine is capable of using control flow commands to execute loops. The environment requests a new command each time the command pointer exceeds the current program. This means that when the agent uses a loop, several previously written commands can be executed in one time step. After each command, the machine updates its internal state accordingly.

In addition to the command pointer, the machine has a stack for the command pointer, a stack for pointers to the input and output data structure, and a skip flag to allow for conditionals.

[2] Example solutions can be found in [9]

We use Python gymnasium to build an environment for the computing machine [7]. The input for our agent is a dictionary of both arrays, both pointer stacks, the program, the last action, the skip flag, the command pointer stack, the command pointer, and the current execution cost.

Command Set There are 21 commands that can be given to the computing machine, which can be divided into the pointer commands, the control flow commands, the conditionals, and the data modifications. The pointer commands include: moving the input pointer right and left, moving the output pointer to the parent and left or right child, and pushing and popping the result pointer to the pointer stack. The control flow commands are *mark:* move the command pointer to the stack; *jump:* jump to the last command pointer on the stack; *drop:* remove the last command pointer on the stack. If a condition is true, we skip the next command, there are the following conditionals: *isnotend*, *isnotstart*, *isnotequal*, *istreeend*, *isnottreestart*, *comparerright*, *leftchildempty*, *rightchildempty*, *nodeempty*. To change the data, you can either *swapright* in the input or *write* the currently selected input to the output.

Action Masking To accelerate learning and to avoid illegal or nonsensical behavior, we employ an action masking strategy. At each timestep, the environment generates a binary mask indicating which actions are valid (mask value = 1) and which are invalid (mask value = 0). If an action is masked, the agent cannot select it. Below are some of the core rules that form the mask:

- The same action cannot be performed twice in a row. For example, if the agent has just jumped, there is little benefit in immediately jumping again.
- We forbid back-to-back conditional commands (e.g., *isnottreestart* followed immediately by *isnotequal*) because the second conditional overrides the skip flag of the first.
- A jump command is only allowed if the previous action was a conditional, so we reduce the risk of endless loops.
- We mask illegal moves, such as moving left when you are already at the leftmost element.
- If the current output array slot is already filled (non-zero) or the agent has already placed that specific number elsewhere, the write action is disallowed.

2.2 Reward Function Design

A key challenge in training a RL agent is to design a reward function that encourages correct behavior. Rewarding partial correctness is particularly difficult. The reward function must effectively handle cases where a node is missing, represented by a 0, and distinguish between small positional errors and large deviations that undermine the BST property. We experimented with two main strategies to quantify the agent's performance: locally: (1) subtree comparison, and (2) edge distance matrix, and globally: (1) independent reward, and (2) incremental reward.

Subtree Comparison: Our initial reward function builds two separate tree data structures, the correct solution tree and a candidate tree derived directly from the environment output array. We recursively compare each pair of matching nodes in the two trees. If both nodes match, no penalty is applied. If one node is missing (or different), a penalty is added, plus any penalty associated with the entire "extra" or "missing" subtree beneath that node. The maximum possible reward is designed to be 0, and the negative number of nodes in the tree is designed to be the minimum possible reward, achieved by an empty array.

Although conceptually simple, this approach overly penalizes small errors. For example, if the agent misplaces or omits the root node, the reward function treats every descendant in that subtree as incorrect, resulting in a large negative sum. This makes it difficult for the agent to receive positive feedback for partially correct placements deeper in the tree. Training instability and slow convergence are common byproducts of this all-or-nothing penalty structure.

Edge Distance Matrix: To combat the excessive negative penalties associated with subtree-based comparison, we introduce a distance-based metric that individually scores the correctness of each node in the final array representation. Again, we represent each BST node by an index i in a level-order array of length n. For a node at index i, the left child is at $2i + 1$, while the right child is at $2i + 2$. The value 0 indicates a missing node and is severely penalized because an omitted node inherently breaks the integrity of the BST.

We then precompute an $edge\_distance\_matrix$ of shape (n, n) as seen in table 1, where each entry $edge\_distance\_matrix[i, j]$ indicates the number of edges one must traverse to get from index i to index j via their lowest common ancestor (LCA). Specifically: For each index i and j, we trace their paths to the root (index 0). The LCA is the first common node in these two paths. The distance is the sum of the distances from i to the LCA and from j to the LCA. To reward correctness more than partial correctness, we assign a large negative shift to the diagonal entries of this matrix, so that placing a node at its own "correct" position is highly rewarded (i.e., avoids a large penalty). During a training episode, the agent outputs a candidate array. We can precompute the matrix because we know the "ideal" index j for each element and the elements do not change their correct positions.

For a value x at index i, the agent receives a reward of $-(\max(edge\_distance\_matrix) + 1)$ if the value at index i is 0 in the agent's solution array (i.e. the agent didn't fill in the value), otherwise it receives a reward of $-edge\_distance\_matrix[j, i]$. This granular scoring scheme prevents a single top-level error from having such a large negative impact on the reward.

In addition to the solution-based reward, we track how many elements in the input array the agent has visited. For each element visited, the agent receives a small positive bonus (e.g., 0.1). This encourages exploration of the array and provides a more densely populated reward function.

To keep the reward signal in a stable range for proximal policy optimization, we normalize the raw negative sum. We first add a $worst\_case$ (i.e.,

$max\_penalty \cdot n$) to shift the cumulative reward into non-negative territory, then divide by the maximum possible sum (which takes into account all correctly present nodes plus the visitation bonus). This results in a final value between 0 and 1, which makes learning more stable.

Global reward Independent of the local reward, i.e. the reward for a step, we need to determine how to reward the agent over the course of an epoch. There are two main approaches, rewarding the state independent of all previous states (independent) or rewarding the agent depending on the improvement over the last step (incremental). The independent reward is easier to compute and simply rewards the agent for the state it is in, but can lead to greedy behavior as the agent is willing to improve the reward quickly to get this reward for all steps. In addition, once the agent has made some progress, exploration becomes less likely, as doing nothing gives a high reward. This reward encourages long programs with a few useful commands in the beginning and then repetition of commands that do not change the result.

On the other hand, the incremental reward, which does not reward actions that do not lead to a better result, makes the reward more sparse, since there will be many commands (such as control flow commands) that do not seem to bring any benefit in the current step.

For both options, the step-based reward is summed over an epoch. The incremental reward is normalized between 0 and 1 if the step reward is normalized in such a way; this is not true for the independent reward, where the epoch reward depends on the length of the program.

Termination reward If the program is able to solve the task, we give it a reward of 1 plus the number of instructions it has left before it reaches the maximum instruction length. This reward is crucial for the independent approach because we end the epoch as soon as the problem is solved, and otherwise the epoch reward would be smaller if the agent solves the task and therefore takes fewer steps. The approach is still relevant for the incremental reward option because we achieve a second goal. It allows the agent to get a higher reward if it solves the task more efficiently. Note that the algorithms are optimized for command efficiency, not execution cost efficiency, which is not the same if the agent uses loops.

2.3 Agent

Our reinforcement learning agent is based on Proximal Policy Optimization (PPO), but we deviate from the standard PPO configuration in important ways. Below we detail the architectural choices we make. We evaluate these choices in section 4.

Default PPO We use the PPO implementation from stable baselines 3 [8]. An overview of the model structure can be found in Figure 1. As mentioned in

$i\backslash j$	0	1	2	3	4	5	6
0	-5	1	1	2	2	2	2
1	–	-5	2	1	1	3	3
2	–	–	-5	3	3	1	1
3	–	–	–	-5	2	4	4
4	–	–	–	–	-5	4	4
5	–	–	–	–	–	-5	2
6	–	–	–	–	–	–	-5

Table 1. Edge distance matrix for a 7-element array-based BST. Diagonal entries (-5) represent a hit and are therefore large negative distances, which later form a positive reward. Dashes (–) indicate omitted cells in the partial upper triangular form.

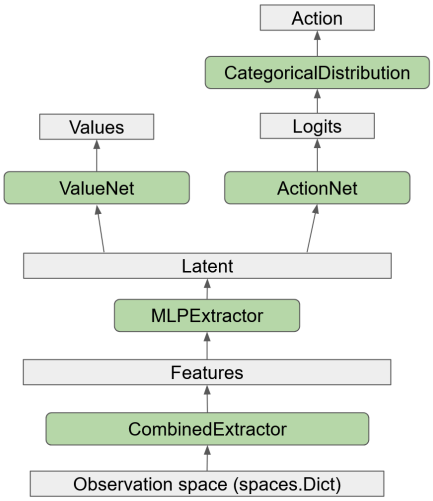

Fig. 1. Default stable baselines 3 PPO architecture

2.1, we return a dictionary of observations. This dictionary is flattened into a single feature vector by the combined extractor, which is then passed to the MLPExtractor, a 2-layer feedforward network with a *tanh* activation function. The state encoding is used by the value net to compute the state value and the action net to compute the logits for the action to play. Both are single-layer feedforward nets. To sample the action to play, the model applies the softmax to the logits to get the probabilities to sample from.

Architectural changes The most important change is the inclusion of an encoder-only transformer [5] to replace the MLPExtractor. We use a transformer because they are better at modeling relational dependencies at scale, and thus we hope to get richer feature embeddings that will help the agent solve the task more easily. Since we only want feature embeddings, we choose a BERT-style [6] encoder-only

transformer. Encoder-only models take the input and output a hidden representation for the input, which can be used as an embedding for the action and value heads.

We choose the Jina AI model "jina-embeddings-v2-small-en" [2] [3]. We specifically choose version 2 over the newer version 3 [3] because the checkpoint is much smaller, with only 33 million parameters. Since we are modifying the MLPExtractor, we need to modify the combined extractor. Our custom combined extractor flattens the observation space dict and includes padding to account for empty slots in the observation. In addition, we add a semantic offset to embed different types of observations differently. For example, input data and pointers have the same value range, so they would be embedded the same. To allow the model to learn embeddings per observation type, we add an offset to remove any overlap in the observation space. Additionally, we add a separation token between observations. We use an attention mask to mask out any padded tokens in the observation space. We replace both the value and action networks with a two-layer feedforward network with a bottleneck size of 128 and a ReLU activation function. Finally, we introduce a new way to control the exploration of the model by allowing temperature to be applied in the softmax of the action net.

2.4 Exploration vs. Exploitation

In initial experiments, we observed that the agent quickly converged to a local minimum and did not explore the action space. Therefore, we systematically evaluate different ways to increase exploration. The PPO loss we are optimizing is given by:

$$Loss_{PPO} = Loss_{policy} + ent\_coef \cdot Loss_{entropy} + vf\_coef \cdot Loss_{value}$$

where the entropy coefficient ent_coef moderates the importance of exploration by penalizing low-entropy (overly deterministic) policies. By increasing this term, we force the agent to explore more.

In addition to the entropy coefficient, we use the temperature we have built into the softmax after the action network. A higher temperature makes the probability distribution smoother and the likelihood of each action closer together. This results in more exploration.

2.5 Active learning approaches

The difficulty of creating a BST and the number of steps required increase with the number of elements in the input data. Therefore, we test whether the agent can transfer knowledge from fewer to more data points. This approach can be compared to active learning approaches in natural language processing, where the context length is increased over the course of training [4]. This is done to help the model contextualize the tokens first locally and then globally. Similarly,

[3] More information about the checkpoint can be found in [10]

we want the model to learn to use the commands and solve the challenge on a small scale in order to use that knowledge to solve it on a larger scale.

As mentioned in Section 2.2, we have two staged rewards, one before the agent solves the task and one after. For the data extension approach, we will focus on the second reward phase, since we want the agent to be able to solve the task before adding more data. We use our evaluation environment as described in section 3 and check at each evaluation epoch whether the agent is able to solve all instances. Only if it solves all, we check if the reward has improved by some delta. If not, we wait the number of epochs defined by our patience and increase the data length when the patience runs out. The check is inspired by the early stopping used in model training, but instead of stopping the training, we change the training environment.

The same reasoning applies to increasing program length during training. By starting with shorter programs, the agent can more often observe the immediate effects of each action and adjust its policy accordingly. If the program is too long, an early error, such as writing a wrong number, can undermine the entire solution, making subsequent actions less effective in shaping the agent's behavior. Therefore, we must allow for a sufficiently large program length to ensure that the resulting policy remains independent of the specific data size. If the program length is too short, the agent simply lacks the capacity to solve the entire problem. We incrementally increase the program length, while also employing an early termination criterion for situations where the reward plateaus. We then increase the data length regardless of whether the agent has fully solved the task.

3 Experimental setup

Hyperparameter search We start our experiments with the default PPO hyperparameters, which can be found in [11]. We also use a fixed seed (42) for all experiments.

We performed a grid search over batch size, learning rate, entropy coefficient, and temperature, resulting in a total of 81 separate training runs. We fix the data length at 7, train for one million steps, and use the LCA-based independent reward. Smaller batch sizes can adapt more quickly to local changes in the reward landscape, but we also include larger batch sizes for more stable policy updates. The learning rate ranged from very small values such as $3 \cdot 10^{-5}$ to moderate values such as $3 \cdot 10^{-3}$. The entropy coefficient was tested at 0.0, 0.1, and 0.5, and the temperature at 1.0, 2.0, and 10.0 to find the best balance between exploration and exploitation.

After evaluating all configurations, the highest episode reward mean was achieved when the batch size was set to 64, the learning rate was set to $3 \cdot 10^{-5}$, the entropy coefficient was set to 0.1, and the temperature was set to 2. This combination achieved both higher final rewards and more stable training dynamics compared to other parameter sets. Although larger batch sizes and higher learning rates converged faster in early iterations, they often plateaued

at suboptimal solutions, while smaller learning rates coupled with moderate exploration allowed the agent to refine its policy more thoroughly. The relatively high temperature and entropy coefficient provided enough exploration for the agent to avoid local optima while still allowing for exploitation.

Evaluation To evaluate the performance of the different configurations, we use an evaluation environment that has the same characteristics as the training environment. Every 10,000 steps the agent performs an evaluation run. The model evaluates five times in the evaluation environment. We record the episode reward, the final result, and the program for the first environment. In the paper we will report the results of the evaluation run with the highest episode reward and the result it produced. We will bold the best results, which does not imply statistical significance, and all correctly placed numbers. Note that we do not enforce deterministic behavior in the evaluation environments.

4 Experimental results

Reward function changes Table 2 shows the different best episode rewards for all reward function and masking experiments. We observe how the change from the subtree based reward to the LCA based independent reward makes a big difference in reward and outcome. We can observe that the agent with the subtree based reward uses shorter episode lengths because each step brings a large negative reward. Instead of trying to learn ways to improve the solution, it tries to find ways to bypass the action masking to bring the machine to an end state. The agent with the LCA-based reward sometimes brings the machine to an illegal state, but much less often in comparison. Nevertheless, the agent tries to get a good result as early as possible, since each subsequent step has a higher reward. We can observe that the agent learns to solve the problem relatively well in the first few steps, accumulating a high reward. After that, the exploration decreases in order not to accidentally decrease the reward. So we end up with solutions that are close to the optimal solution. Increasing the reward for correct solutions did not solve the problem.

The incremental reward is the only change to the basis that causes the agent to solve the task. Although the subtree based reward underperforms when the reward is calculated independently, we observe that both local reward calculations can lead to correct solutions when used incrementally. The LCA based reward still outperforms the subtree based reward because the agent only needs 22 instead of 33 commands to solve the problem.

In the incremental reward, we do not care which step the improvement comes from, while in the independent reward, an early improvement is weighted much more than a late improvement. Thus, the incremental reward solves the greedy nature of the independent reward. The greedy behavior causes the reward rules to fall apart, e.g. it might be advantageous to place a false node early rather than correct it later if the program does not solve the problem. Thus, the agent greedily descends to a local maximum at the beginning of training. As a side

effect, the incremental reward removes the length penalty in the subtree based reward. The first step still gets a negative reward, but the following steps get a positive reward if they really improve the solution. This makes the episode lengths comparable.

Interestingly, despite these differences, all agents learned that the first action should be to go into the left child, with a high probability after 200,000 steps. Both solutions that solve the problem agree on the first 3 commands and use the same strategy: They go through the array from left to right and insert the numbers into the BST in order.

The action masking experiments also can be found in table 2. Action masking is important for our model because it limits the search space of the algorithms. One of these problems can be found in the result of no action masking: the agent writes the 6 twice. This cannot be a good result because our input never contains a number twice. With action masking, the algorithm does not have to learn this context and can focus on learning the task.

We observed that our agent does not use loops and other control flow commands. Therefore, we test a naive approach where only *write, left child, right child,* and *parent* are unmasked to remove most of the search space. We do not add any more action masking. Although we have reduced the search space, the agent is still unable to solve the task. Just as with no action masking, the agent does not learn that each number should be written only once. Even with the well-performing incremental reward, the agent does not solve the problem. These findings show, the importance of action masking compared to command pruning.

	Reward	Result
Independent LCA reward (base)	45.747	[**4 2 6** 0 **3 5 7**]
Independent subtree reward	-45.000 (in new reward: 0.571)	[0 0 0 **1** 0 0 0]
Incremental LCA reward	43.000	[**4 2 6 1 3 5 7**]
Incremental subtree reward	29.000	[**4 2 6 1 3 5 7**]
Base, no action mask	27.300	[**4** 0 **6** 0 0 6 **7**]
Naive	35.079	[**4 2 6** 7 7 7 **7**]
Naive and incremental LCA reward	0.8713	[**4 2 6** 7 **3 5 7**]

Table 2. The best episode reward and the result based on different reward functions and action masking. We did not bold the reward as the old, new and incremental reward are not comparable.

Entropy regularization In our sweep, we found that the run with an entropy coefficient of 0.1 and a temperature of 2.0 performed best. Nevertheless, we want to highlight the effect of these entropy regularizations in the table 3. In cases where both the temperature and the entropy coefficient were set unusually high, the agent showed constant exploration without settling on a stable solution strategy. By overemphasizing randomness in the action selection process, the policy

failed to maintain favorable behavior across update cycles, effectively "forgetting" progress on partial solutions. As a result, it repeatedly entered unprofitable action sequences and failed to converge to a consistent BST construction method.

Adding the small regularization improves the payoff, although the results without regularization can be considered equally good, since they are both one edit away from the solution. Interestingly, the agent underperforms when only the entropy coefficient is added without temperature. This could be related to the fact that the agent has a loss signal to explore more, but is not able to translate it into meaningful actions.

Entropy coef	Temperature	Reward	Result
0.0	1.0	42.726	[**4** 1 **6** 2 **3 5 7**]
0.0	2.0	43.105	[**4 2** 5 **1 3** 6 **7**]
0.1	1.0	38.686	[**4 2 6** 0 0 **5 7**]
0.1	**2.0**	**45.747**	[**4 2 6** 0 **3 5 7**]
0.1	10.0	23.976	[1 **2 6** 0 0 4 **7**]
0.5	1.0	26.941	[**4 2** 5 0 0 0 **7**]

Table 3. The best episode reward and the result based on different values for the entropy coefficient and the temperature.

Architectural experiments The default PPO model from stable baselines is quite small, so we made architectural changes (see 2.3). Table 4 shows that the base variant with all architectural changes performs best. We do not observe much change in the result when initializing the encoder model from scratch, which makes sense since we have to reinitialize the embedding matrix anyway, and our state tokens do not represent any lingual tokens from pre-training. The action network does not need to be increased, as the results are equivalent in quality, but the added parameters are much fewer compared to the parameters added by the encoder. In contrast, the inclusion of a larger value net dramatically improves performance.

Interestingly, the default encoder performs similarly to the Transformer encoder. We have tested different approaches to increase the size of the encoder, but keep it as just an MLP without attention. These tested models decreased performance. The MLP, which is about the same size as the Transformer encoder, consists of 16 linear layers with a hidden size of 1024 and a tanh activation function, it performs worse than the default encoder, showing that we do not need more parameters, we need better modeling. Better modeling includes newer activation functions and the use of attention to improve the contextualization of the machine state.

We hypothesize that a problem of the larger models is the independent reward function. The reward function prioritizes greedy action choices, and the agent changes little after 200,000 steps, so the larger models do not have enough data

to improve the encodings before the model overfits. With a different reward function, such as incremental reward, we could see larger effects of the encoder models. Especially since the current agent does not use control flow arguments, which would require more notion of commands.

	Reward	Result
Base	45.747	[**4 2 6** 0 **3 5 7**]
not pretrained	42.693	[**4 2** 5 **1 3** 6 **7**]
same size MLP	36.96	[**4 2 6** 0 **3** 0 **7**]
default encoder	45.153	[**4 2 6 1 3** 0 **7**]
default action net	45.576	[**4 2 6 1 3** 0 **7**]
default value net	38.972	[**4 2 6** 0 0 **5 7**]
default PPO	26.221	[**2** 0 **6** 0 0 **5 7**]

Table 4. The best episode reward and the result based on different architectural changes.

Table 5 shows the results for the active learning approaches. The rewards look much worse compared to the training without active learning, but the independent reward depends on the episode length, and we do not increase the program length to the lengths available in the base case. Therefore, we calculated the reward per maximum episode length. With this and the results we observe that increasing the program length does not hurt or improve performance, the results are comparable. We hypothesize that this gradual increase has little effect, since the greedy reward optimization already optimizes the first steps in the beginning. Increasing the program length could actually strengthen this greedy approach by leading to local optimization of the first steps.

Increasing the data length gradually does not work well. The agent solves the problem for 2 data points, but until we switch to 3 it is overfitted on 2, so it is not able to learn it for 3. As seen in the result, the agent still predicts the solution for 2. Increasing both at the same time does not work.

5 Conclusion

We investigated how a reinforcement learning agent can transform sorted arrays into valid Binary Search Trees by learning a sequence of low-level commands. Our results suggest that a few specific design choices play a critical role in enabling the agent to construct correct BSTs. In particular, the incremental reward function proved essential for encouraging continuous exploration and avoiding premature termination. In addition, increasing the value head yielded richer feature representations than smaller multilayer perceptrons, facilitating more reliable policy learning. We also found that introducing a custom reward function based on the distance matrix significantly improved the agent's ability to correctly place nodes in the BST array. Finally, action masking emerged as

	Reward	Reward per maximum episode length	Result
Base	45.747	0.726	[**4 2 6** 0 **3 5 7**]
increase data length	16.667	0.265	[**2 1** 0]
increase program length	28.006	0.667	[**4 2 6 1 3** 0 **7**]
increase both	8.512	0.473	[**2** 0]

Table 5. The best episode reward and the result based on different active learning approaches. Note that the rewards for the runs with less program length are smaller, because more rewards can be accumulated in more steps without the final result being better. Therefore, the rewards are not directly comparable and we do not bold them. In addition, we compute the reward per maximum number of steps available at the end of training.

another important factor that streamlined exploration by filtering out invalid or redundant actions.

Custom action networks, entropy coefficient adjustments, and active learning approaches had minimal or negative effects on final performance. Although small variations in entropy or network architecture occasionally influenced the dynamics of early training, these components did not consistently improve reward or BST construction accuracy.

Overall, our experiments suggest that the combination of incremental rewards, a well-structured reward function, and targeted action masking is critical in guiding the RL agent to learn a functional BST construction policy. We found that these environment level hyperparameters were more influential than changes to the PPO model.

6 Limitations

Although our results are promising, several limitations limit the generalizability and comprehensiveness of this study. First, only one transformer architecture was evaluated, leaving open the question of whether different self-attention models or lighter encoders might perform as well or better. Second, we tested only one data length for the input arrays, so it remains uncertain how well the agent's learned policy scales to significantly larger or smaller data sets. Since we give the agent the arrays sorted, this means that the agent only learns on one example and most likely overfits on it, as seen when trying to increase the data length. Third, our experiments did not exceed one million training steps, so the performance of the RL agent at higher scales, where more extensive exploration and fine-tuning could take place, remains unknown. In addition, the hyperparameter tuning focused on a limited set of variables, suggesting that further exploration of learning rates, batch sizes, and exploration parameters may yield better results. Although we conducted a thorough grid search over several hyperparameters, our experiments were run with a single random seed. This choice provided a first indication of the agent's performance, but may not capture the variance across different seeds. As a result, the reported performance and behavior may

not generalize equally to other random initializations. Finally, alternative observation spaces that differ in the representation of pointer stacks and arrays could radically alter the agent's learning process and potentially improve performance. An example of such different observation spaces can be found in [1], where they use one-hot encodings to encode the observation space.

7 Future work

Based on these limitations, there are several ways to extend and improve this approach. First, testing different encoders, such as different transformer variants or graph-based architectures, may reveal more efficient ways to capture structural relationships in BST construction. Second, exploring alternative observation spaces may provide new insights into how best to represent pointers, arrays, and tree structures for RL-based synthesis. It would be interesting to evaluate whether our relatively large observation space improves the performance of the model. A minimal observation space may only provide the data length and the current program.

A logical next step would be to rerun all experiments with incremental rewards, as we see the biggest positive impact from this change. With the incremental reward, the agent may be able to solve larger inputs.

In addition to scaling experiments beyond one million steps and exploring different architectures, a critical next step is to repeat all experiments with multiple seeds to better assess the stability and reproducibility of our results. By evaluating performance across multiple random initializations, we can draw more robust conclusions about the agent's generalization abilities and eliminate the possibility of seed-specific biases. Finally, introducing additional incentives for loops and other control flow instructions may encourage the agent to discover more compact or efficient programs for BST construction, bringing the learned strategies closer to state-of-the-art or even novel algorithmic strategies.

References

1. Mankowitz, D., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., and Paduraru, C., Leurent, E., Iqbal, S., Lespiau, J., Ahern, A. and others: Faster sorting algorithms discovered using deep reinforcement learning. Nature **618**(7964), 257–263 (2023)
2. Günther, M., Ong, J., Mohr, I., Abdessalem, A., Abel, T., Akram, M. K., Guzman, S., Mastrapas, G., Sturua, S., Wang, B., Werk, M., Wang, N., and Xiao, H.: Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents. arXiv preprint arXiv:2310.19923 (2024)
3. Sturua, S., Mohr, I., Akram, M. K., Günther, M., Wang, B., Krimmel, M., Wang, F., Mastrapas, G., Koukounas, A., Wang, N., and Xiao, H.: jina-embeddings-v3: Multilingual Embeddings With Task LoRA. arXiv preprint arXiv:2409.10173 (2024)
4. Nagatsuka, K., Broni-Bediako, C., and Atsumi, M.: Pre-training a BERT with curriculum learning by increasing block-size of input text. In Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2021), pages 989–996 (2021)

5. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I.: Attention Is All You Need. In Advances in Neural Information Processing Systems, edited by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Vol. 30 (2017)
6. Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K.: BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), edited by Jill Burstein, Christy Doran, and Thamar Solorio, pages 4171–4186 (2019
7. Gymnasium Homepage, https://gymnasium.farama.org/index.html, last accessed 2025/03/09
8. Stable Baselines 3 PPO, https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html, last accessed 2025/03/09
9. Leetcode: Convert Sorted List to Binary Search Tree, https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/, last accessed 2025/03/09
10. Hugging Face: Model card jina-embeddings-v2-small-en, https://huggingface.co/jinaai/jina-embeddings-v2-small-en, last accessed 2025/03/09
11. Default PPO hyper-parameters: https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/ppo/ppo.py last accessed 2025/03/09