ACR-Tree: Constructing R-Trees Using Deep Reinforcement Learning

Shuai Huang, Yong Wang, and Guoliang Li(✉)

Tsinghua University, Beijing, China
{huang-s19,wangy18}@mails.tsinghua.edu.cn, liguoliang@tsinghua.edu.cn

Abstract. The performance of an R-tree mostly depends on how it is built (how to pack tree nodes), which is an NP-hard problem. The existing R-tree building algorithms use either heuristic or greedy strategy to perform node packing and mainly have 2 limitations: (1) They greedily optimize the short-term but not the overall tree costs. (2) They enforce full-packing of each node. These both limit the built tree structure. To address these limitations, we propose ACR-tree, an R-tree building algorithm based on deep reinforcement learning. To optimize the long-term tree costs, we design a tree Markov decision process to model the R-tree construction. To effectively explore the huge searching space of non-full R-tree packing, we utilize the Actor-Critic algorithm and design a deep neural network model to capture spatial data distribution for estimating the long-term tree costs and making node packing decisions. We also propose a bottom-up method to efficiently train the model. Extensive experiments on real-world datasets show that the ACR-tree significantly outperforms existing R-trees.

Keywords: R-tree · Reinforcement learning · Spatial index

1 Introduction

Querying spatial objects is fundamental in location-based applications such as map services and social networking. For example, an urban resident may search points of interest (POIs) such as restaurants in some region (range queries) specified on Google Maps or sometimes query the k-nearest POIs (kNN queries). R-trees [5] are adopted to index these spatial objects and speed up spatial queries, especially when there are hundreds of millions of objects or a huge number of queries.

Given a set of spatial objects, an R-tree can be constructed by incrementally inserting each object [2,3,5,16]. However, this approach is not efficient for building an R-tree from scratch and often leads to poor R-tree structure with bad query performance. Therefore, there are approaches attempting to build more efficient R-trees in a bulk-loading manner. There are two types of strategies: (1) Bottom-up methods [1,6,8,12,14] pack objects into parent nodes recursively, based on hand-crafted heuristics. (2) Top-down method [4] partitions a node into

X. Wang et al. (Eds.): DASFAA 2023, LNCS 13943, pp. 80–96, 2023.
https://doi.org/10.1007/978-3-031-30637-2_6

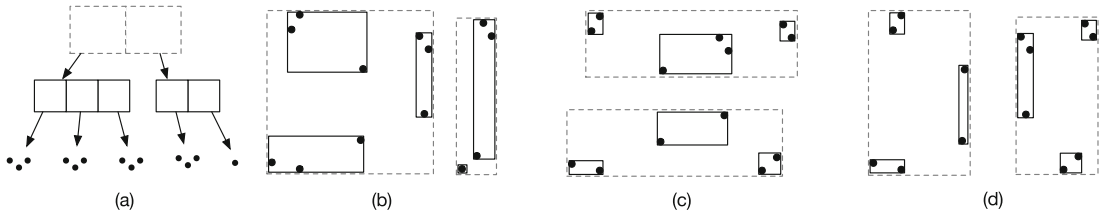

Fig. 1. R-tree constructed under various settings (the MBR of nodes in the higher level is represented by rectangles with dashed edges and lower level with solid edges). (a) (b): under full-node-packing constraint, (c): allowing packing of non-full nodes, (d): considering long-term tree costs.

child nodes recursively, greedily optimizing each tree construction step. These approaches try to achieve evenly distributed objects in different subtrees and full packing of entries of each subtree.

Limitations of the Existing Methods. (**L1**) They omit the long-term results (the whole node partition or the overall tree construction) when making the decision in each step. It may lead to a bad overall result. (**L2**) They impose an unnecessary full-packing constraint, *i.e.,* full-filling each tree node with entries to its capacity, on the tree construction. It limits the resulting tree structure.

For example, Fig. 1 shows 3 cases when we build a 3-level R-tree on a set of spatial object data, and we want to minimize the total area of minimum bounding rectangle (MBR) of tree nodes which mainly affects the query performance. If we build it by [4] under full-packing constraint, we will get the R-tree shown in (a) (b). We can observe that data points which are far from each other are packed together, resulting in nodes with big MBRs. As a comparison, if we allow a variable number of entries in each node, we can build an R-tree as shown in (c), where the MBR areas are significantly smaller. Furthermore, since (c) greedily optimizes the areas in the current level, we can further improve it by considering the areas in lower levels. In this way, we can build an R-tree in (d) of which the upper-level nodes are a bit larger than (c) but nodes in the lower level are significantly smaller thus the overall result is better.

Challenges and Our Proposed Solutions. To address these limitations and build R-trees with better query performance, we need to search in a much larger space, optimizing the long-term tree costs, and the traditional methods are impractical, *i.e.,* unable to enumerate all the possibilities. Therefore, we propose to build R-trees using deep reinforcement learning (DRL), which has been proven successful in applications of database systems such as configuration parameters tuning [13] and join order selection [18]. RL can learn to optimize the overall benefits of a complex task through trial-and-error explorations. It typically incorporates the generalization ability of deep learning (DRL), which allows us to effectively learn a strategy from large search space with limited explorations.

However, there are several challenges in utilizing DRL in R-tree building: (**C1**) How to model the R-tree building process and optimize the long-term tree costs (**L1**)? We propose a tree Markov decision process (tree-MDP), which

allows us to optimize the overall tree costs (*i.e.,* considering the rest steps and rest levels). (**C2**) The searching space to build an R-tree is huge, which becomes much larger when we remove the full-packing constraint (**L2**). How to effectively explore good solutions? We utilize Actor-Critic, a DRL algorithm using neural networks to automatically learn the R-tree building strategy from limited explored samples. We design a grid-based model to encode the spatial distribution of data and use a hierarchical convolutional network to embed it into a vector, for effectively estimating the long-term tree costs (Critic) and making R-tree building decisions (Actor). (**C3**) It's time-consuming to collect feedback (*i.e.,* R-tree building experiences) and train the multi-level Actor-Critic models. We design a bottom-up model training framework where we use a training-sharing method to reduce training rounds and a shortcut method to efficiently access feedback in each round. Our main contributions are summarized as follows.

- We propose to utilize DRL to address 2 limitations in the existing R-tree batch-building algorithms, *i.e.,* greedy strategy and full-packing constraint.
- To model the top-down R-tree construction, we propose tree-MDP which well fits the recursive R-tree building process and models the long-term tree costs.
- We use Actor-Critic networks to effectively explore the huge searching space of R-tree construction. We design a grid-based representation model and a hierarchical network to embed the spatial distribution of data. Then we can effectively estimate the long-term tree costs and make building decisions.
- We propose a bottom-up model training framework, where we use training-sharing and shortcut methods to accelerate model learning.
- We have conducted extensive experiments on real-world datasets and the results show that our method outperforms existing approaches by 20%–30%.

2 Related Work

2.1 Spatial Queries and R-Tree

We study spatial queries on multi-dimensional objects such as points, rectangles and polygons. Given a set of n d-dimensional objects, we consider two common types of spatial queries: *range (or window) query* and *k-nearest neighbor (kNN) query*. A range query retrieves all the objects that are included by or intersect a rectangular range. A kNN query retrieves k closest objects to a given coordinate p in Euclidean space.

To avoid scanning all objects to answer a query, spatial indices especially R-trees are proposed for efficient searching.

R-Tree. The R-tree [5] is a balanced tree structure for indexing spatial objects and is widely used. In an R-tree, Each tree node N contains up to B entries, where B is determined by the disk block size. Therefore, the height H of an R-tree indexing n objects is lower bounded by $O(\log_B n)$. Each entry in a node contains a pointer to an child node (or object) and the minimum bounding rectangle (MBR) that surrounds it.

An R-tree greatly speeds up spatial queries by traversing tree nodes in a top-down manner, during which subtrees whose MBRs do not intersect a range query rectangle or are not promising to be in results of a kNN query can be safely pruned.

Extensive studies [2,3,5,7,16] design heuristics to optimize the query performance during dynamic data insertion. For example, the R*-tree [2] minimizes the areas, overlaps and perimeters of node MBRs when choosing the node to insert a new object, so that less tree nodes are expected to be accessed in queries.

R-Tree Building. When building an R-tree from scratch, instead of one-by-one insertion, batch building (packing) methods are designed for reducing construction costs and achieving better tree structure. Existing methods consider full packing, *i.e.,* filling each node to its capacity if possible, with a 100% space utilization.

Most of these methods take a bottom-up manner, *i.e.,* packing every B objects into one parent node, recursively until reaching the root (less than B child nodes). For example, [8] applies Z-order and Hilbert-order on multi-dimensional spatial objects, then sequentially packs each B ones. [14] adds a further step, *i.e.,* mapping the spatial points into a rank space, before sorting and packing them, which guarantees a theoretical worst case bound on the query performance. STR [12] first divides the objects into $\sqrt{n/B}$ groups by the order of their x-coordinates, then packs each B objects in each group by y-coordinates. PR-tree [1] provides a worst-case query cost, which is asymptotically optimal, by a recursive 6-partitioning process. These heuristic-based methods rely on the uniformity of data distribution and can not achieve good performance on many real-world datasets with skewness [4].

There are also top-down methods, *i.e.,* partitioning a parent node into up to B child nodes, recursively until no more than B objects in each as the leaf nodes. TGS [4] uses a greedy partitioning strategy: repeatedly splitting the objects into 2 subsets, by a cut orthogonal to one of the d axes. TGS optimizes the area sum of the 2 MBRs of resulting subsets, by enumerating all the candidate cuts. TGS has better performance than others on most datasets. However, the greedy strategy omits the long-term tree costs and may have a bad overall result.

Moreover, all these methods are limited by the full-packing constraint which reduces the problem space but limits the built R-tree structure. Different from them, we remove this constraint and utilize deep reinforcement learning to effectively optimize the overall R-tree costs.

2.2 Deep Reinforcement Learning

Reinforcement learning (RL) is a powerful algorithm that can learn to make decisions in a complex task (*e.g.,* chess [17]), maximizing specific benefits, through trial-and-error exploration. It has also been successfully applied to problems in databases such as knob tuning [13] and join order selection [18].

The task (*e.g.,* building an R-tree) is usually divided into multiple steps (*e.g.,* node packing). In each step, the decision maker, *a.k.a.agent*, based on the current *state* (*e.g.,* the spatial distribution of data) takes an *action* (*e.g.,* packing a node),

then gets a *reward* (*e.g.,* cost of the node) and moves to the next step. This is the Markov decision process (MDP) into which we need to formulate the task in RL. The *agent* learns a *policy*, *i.e.,* action selection strategy, that maximizes the long-term *rewards* (*e.g.,* the total R-tree cost), through the exploration experiences. When the task has a large searching space, RL usually incorporates deep learning (DRL) of which the generalization ability enables it to effectively learn the *policy* from limited exploration.

Actor-Critic is a class of DRL methods which is wildly used. It uses a deep neural network (DNN) model to represent the *policy* (Actor) which can make decisions (*action*). It uses another DNN model to estimate the long-term *rewards* (Critic) which can help improving the Actor. PPO [15] is a popular policy gradient method, which is a default choice at OpenAI[1], that updates the policy (*i.e.,* Actor) through a "surrogate" objective function. Our method utilizes the Actor-Critic framework and uses PPO to learn the Actor model.

3 Framework

We build an R-tree in a top-down manner because tree nodes closer to the root have larger impact on query performance, which are better to be considered first [4]. Specifically, first, we divide the n objects into $x \leq B$ groups, and each group corresponds to a child node of the root node, *i.e.,* partitioning the root node. Next, it recursively partitions these child nodes until every node has no more than B objects which become the leaf nodes. The key point is how to partition each tree node. We first split the object set into 2 subsets by a cut orthogonal to an axis, then recursively split the resulting subsets and finally get x child nodes.

The main challenge in this R-tree building process is that the search space of node partition (by multiple splitting operations) is huge. To make it practical and efficient to implement, the traditional top-down method TGS [4] takes 2 settings:

Full-Packing Constraint. TGS enforces packing full-filled child nodes, *i.e.,* a node of abundant objects will be partitioned into the least possible nodes, with each node except the last one filled up to its capacity. Under this constraint, the possible choices of each split are limited to $O(B)$, instead of $O(n)$ when without this constraint.

Greedy Split. TGS only considers how to optimize the current split operation, *i.e.,* from the $O(B)$ candidates choosing the "best" split that minimizes the area sum of the MBRs of 2 resulting subsets, which may be further split.

However, these 2 settings limit the R-tree building results as Sect. 1 and Fig. 1 show. To overcome these 2 limitations and search a better R-tree structure from the larger space, we utilize Actor-Critic [9], a DRL algorithm and propose **ACR-tree** (Actor-Critic R-tree), of which the framework is shown in Fig. 2. We use tree-MDP (**M1**, Sect. 4.1, 4.2) to model the top-down R-tree building process, which provides a framework to optimize the long-term tree costs. We design the

[1] https://openai.com/.

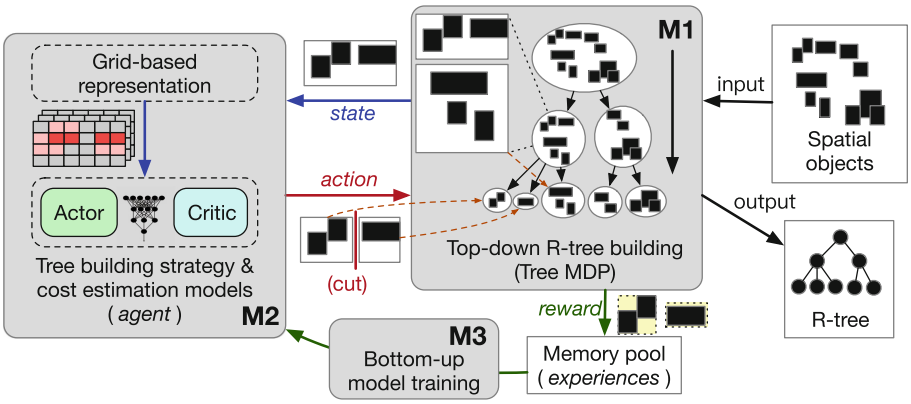

Fig. 2. ACR-tree building framework

Actor-Critic networks as the *agent* (**M2**, Sect. 4.3) to make decisions in each building step in the tree-MDP framework.

Now we introduce the workflow of building an ACR-tree. Given a set of spatial objects, we build an R-tree through top-down node partitioning, beginning at the root node containing all the objects. Each node partitioning is achieved by a recursive object set splitting process consisting of multiple steps. In each step, to split an object set, the spatial distribution of the objects is passed as the *state* to the Actor-Critic *agent*. The *agent* first uses a grid-based model to encode the spatial *state*, then uses a neural network model to embed the spatial features and generates an *action*. The *action* can be a splitting operation, *i.e.,* using a cut orthogonal to an axis to split the objects into 2 subsets. It can also be a packing operation, *i.e.,* stopping splitting the current set which can fit in one child node and packing it, and then getting a *reward* (*e.g.,* area of the node MBR). When all the nodes need no more partition, we built an R-tree.

To train the *agent* model, we repeatedly perform the above R-tree construction process and store the *rewards* into a memory pool, and use them as the feedback to update the parameters of both Actor and Critic models. We need to train several Actor-Critic pairs, one for each level. The training is time-consuming and we propose a bottom-up framework (**M3**, Sect. 5) for efficient model training.

4 Actor-Critic R-Tree

In this section, we first specifically show the R-tree building process without full-packing constraint, which consists of multiple splitting operations (Sect. 4.1). We then introduce the tree-MDP to model it (Sect. 4.2). Finally, we introduce the Actor-Critic networks, which make the splitting decisions in the process (Sect. 4.3).

4.1 Top-Down R-Tree Building Process

Algorithm 1 shows the top-down R-tree building framework, which is similar to TGS [4]. Partitioning the objects $\mathcal{O}$ contained in a node into at most B subsets (*i.e.,* child nodes) (line 5) is the key point in this framework, which decides the

Algorithm 1: CreateTree (Top-down R-tree Building)

Input: $\mathcal{O}$: n objects data, B: node capacity
1 **if** $n \leq B$ **then**
2 | $entries = \{\langle MBR(\{o\}), o\rangle | o \in \mathcal{O}\}$
3 **else**
4 | $entries = \{\}$
5 | $objectSubsets = NodePartition(\mathcal{O}, B)$
6 | **foreach** $\mathcal{O}_{ch} \in objectSubsets$ **do**
7 | | $N_{ch} = CreateTree\ (\mathcal{O}_{ch},\ B)$
8 | | Add $\langle MBR(\mathcal{O}_{ch}), N_{ch}\rangle$ into $entries$
9 **return** *A pointer to entries*

Algorithm 2: NodePartition

Input: $\mathcal{O}$: n objects, P: max nodes, H: node height ($\lceil \log_B n \rceil$ by default)
Output: Object subsets contained by each child nodes
1 **if** $n \leq B^{H-1}$ *and* ($P = 1$ *or* $Pack(\mathcal{O}, P, H)$) **then**
2 | **return** $\{\mathcal{O}\}$
3 $dim, pos, \alpha = Cut(\mathcal{O}, P, H)$
4 sort $\mathcal{O}$ by the upper coordinates on dim, n_l of which are below pos
5 Adjust n_l to satisfy $0 < n_l < n$ and $\lceil \frac{n_l}{B^{H-1}} \rceil + \lceil \frac{n-n_l}{B^{H-1}} \rceil \leq P$
6 Take the first n_l ones from $\mathcal{O}$ as $\mathcal{O}_l$, the rest as $\mathcal{O}_r$
7 $P_l^{min} = \lceil \frac{|\mathcal{O}_l|}{B^{H-1}} \rceil, P_r^{min} = \lceil \frac{|\mathcal{O}_r|}{B^{H-1}} \rceil,\ P_{extra} = P - P_l^{min} - P_r^{min}$
8 $P_l = P_l^{min} + \lfloor \alpha P_{extra} \rfloor, P_r = P - P_l$
9 **return** $NodePartition(\mathcal{O}_l, P_l, H) \cup NodePartition(\mathcal{O}_r, P_r, H)$

final tree structure. Different from TGS, in our ACR-tree, *NodePartition* allows the child nodes to be not full packed, *i.e.,* for any $\mathcal{O}_{ch} \in objectSubsets$, $|\mathcal{O}_{ch}|$ can be arbitrary but not exceeding B^{H-1} where $H = \lceil \log_B |\mathcal{O}| \rceil$.

Algorithm 2 show the recursive *NodePartition* based on set splitting. P is the maximum number of nodes to be finally packed ($P = B$ at the beginning). We do not enumerate all possible splits but choose a cut orthogonal to 1 of the d dimensions to divide the object set (line 3). *e.g.,* When choose the X-*dim*, it's a vertical line $x = pos$. We also specify a ratio α to divide the available nodes P into P_l, P_r (line 7, 8) for the 2 resulting subsets $\mathcal{O}_l$ and $\mathcal{O}_r$ respectively. Then we call *NodePartition* on $\mathcal{O}_l$ and $\mathcal{O}_r$ recursively to further split them, and merge the results as the final set of child nodes (line 9). When the objects can fit in one child node ($n \leq B^{H-1}$), we are forced to stop splitting and pack them if the maximum node number $P = 1$, else we can choose whether to pack (line 1).

Example 1. Figure 3 shows the construction of a 3-level R-tree when $B = 3$. We first partition the objects $\{o_1 - o_{11}\}$ inside the root node. We first use a cut $x = c_0$ (yellow dash line) to split the objects into 2 subsets $\{o_6 - o_{11}\}, \{o_1 - o_5\}$, and use ratio $\alpha = 0.5$ to divide the maximum node number $P_0 = 3$ into $P_1 = 1, P2 = 2$. Then we are forced to pack $\{o_6 - o_{11}\}$ as N_1 since $P_1 = 1$. We also choose to pack $\{o_1 - o_5\}$ as N_2. Similarly, we can partition N_1, N_2 in level 2 and get $N_3 - N_7$, all of which contain no more than 3 objects, and we finally get an R-tree.

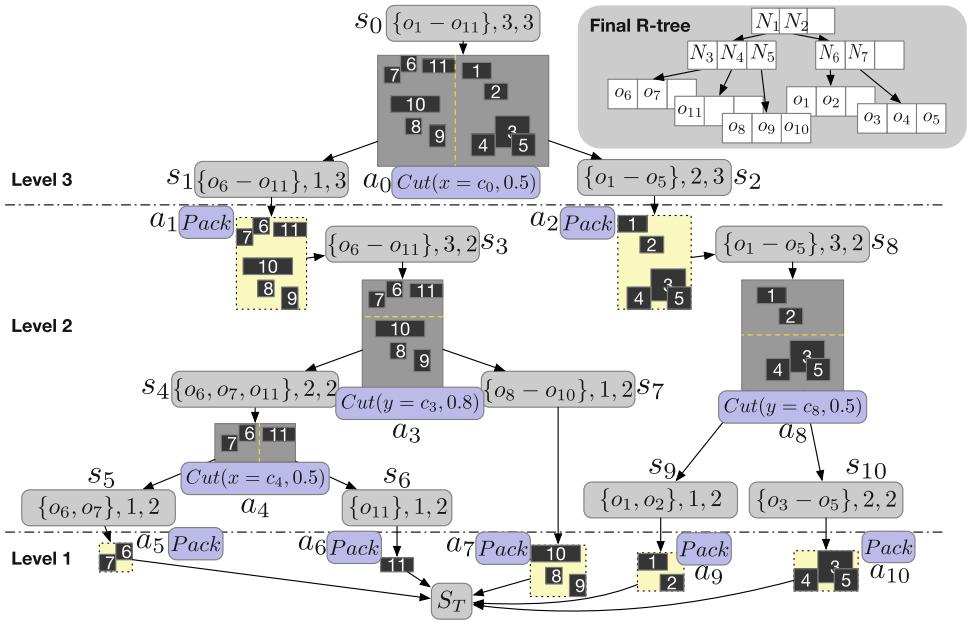

Fig. 3. An example of top-down R-tree construction when $B = 3$

The selection of *Cut* and *Pack* in each step will influence the future splitting and the final partitioning result. We utilize DRL to learn to make these splitting decisions that minimize the long-term R-tree costs. We first need to model the R-tree building process as an MDP.

4.2 Tree MDP Model

An MDP is usually sequential and chain-like, where we move from one state to another each time we make a decision. For example, a state can be the current board in a chess game [17]. However, the construction process of an R-tree is tree-like and the general MDP is hard to model it. Therefore, we propose tree-MDP, where a state (*i.e.,* an object set to be split) can have several succeeding states (*i.e.,* the resulting 2 subsets), and the long-term rewards of a state also depend on the succeeding actions and states only. Next, we specifically define the tree-MDP model (also use Fig. 3 for explanation).

State. A state s_t is a set $\mathcal{O}$ of rectangles, with a maximum node number P and a node height H. *e.g.,* the construction begins at $s_0 = \langle \{o_1 - o_{11}\}, 3, 3 \rangle$.

Action. There are 2 types of action:

- *Pack*: Packing the current objects into a new child node, then (a) if $H = 2$ (leaf node), move to the terminal state S_T, (b) else move to a new state $s_{t'}$ of a non-leaf node with $P' = B, H' = H - 1$, which need to be further partitioned.
- *Cut*: Using a cut (dim, pos, α) to split $\mathcal{O}$ and P into 2 succeeding states s_{t_l}, s_{t_r}.

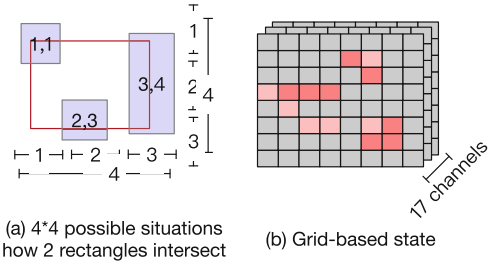

Fig. 4. Grid-based representation model

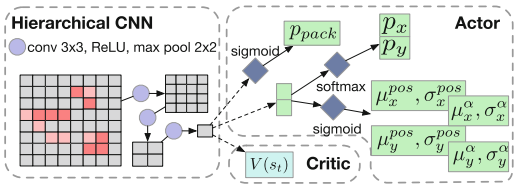

Fig. 5. Hierarchical Actor-Critic network

For example, the action a_0 on s_0 specifies a cut ($x = c_0$, the yellow dash line) to divide $\mathcal{O}$ into $\{o_6 - o_{11}\}$ and $\{o_1 - o_5\}$, and a ratio $\alpha_0 = 0.5$ to divide $P_0 = 3$ into $P_1 = 1, P_2 = 2$, into 2 succeeding states s_1, s_2 respectively. The action a_1 on s_1 packs the current object set where $|\{o_6 - o_{11}\}| \leq 3^2$ as a child node N_1.

A policy π is a mapping from states to the action distribution, *i.e.*, $a_t \sim \pi(s_t)$.

Reward. To optimize the overall R-tree cost (*i.e.*, node areas), we define the reward $r_t = r(s_t, a_t)$ as: (1) If a_t is *Cut*, then $r_t = 0$; (2) If a_t is *Pack* yielding a new child node N, then $r_t = area(N)$. *e.g.*, $r_0 = 0$, $r_1 = -area(N_1)$.

We use a **value function** $V_\pi(s_t)$ to evaluate the long-term reward, *i.e.*, rewards of all the successors (subtree) of state s_t, under specific policy π. For terminal state $V_\pi(S_T) = 0$, else the value can be recursively defined as:

$$V_\pi(s_t) = \mathbb{E}_{a_t \sim \pi(s_t)} \begin{cases} r(s_t, a_t) + \gamma_1 \left[V_\pi(s_{t_l}) + V_\pi(s_{t_r}) \right] & , a_t \text{ is } Cut \\ r(s_t, a_t) + \gamma_2 V_\pi(s_{t'}) & , a_t \text{ is } Pack \end{cases} \tag{1}$$

where γ_1, γ_2 are 2 discounting factors (*e.g.*, 0.99) for better convergence.

We can observe that $V_\pi(s_t)$ is the discounted sum of node areas in the subtree from s_t. Therefore, we need to choose an action maximizing $V_\pi(s_t)$. However, in this tree construction process without full-packing constraint, the search space is huge and we can only explore a small part. Therefore a traditional search-based method (*e.g.*, Monte Carlo tree search) may yield a bad result. Therefore, we utilize the Actor-Critic algorithm, which uses neural network models to learn from limited samples and generalizes to the unexplored space, thus is effective for large searching space.

4.3 Actor-Critic Model

In order to make good decisions in the tree-MDP, the agent learns 2 models, *i.e.*, Actor to generate partitioning actions and Critic to estimate the long-term R-tree costs. Note that only Actor is not enough and we need Critic to assist in improving it. Moreover, a trained Critic model can also be utilized to accelerate the training process, which will be introduced in Sect. 5.2.

Since the policies and value functions in various tree levels are different, we learn $H-1$ models ($h = 2, ..., H$) for partitioning nodes of various object set sizes, *i.e.*, model h for partitioning $n \in (B^{h-1}, B^h]$ objects. Both of Actor and Critic

in level h take a state $s_t = \langle \mathcal{O}_t, P_t, H_t \rangle$ (where $H_t = h$ is constant) as input, then Actor generates $a_t \sim \pi(s_t)$ and Critic estimates $V_\pi(s_t)$. The performance strongly depends on the effectiveness of embedding the spatial distribution of the rectangle set $\mathcal{O}_t$. First, we use a grid-based model to represent $\mathcal{O}_t$, which can be better captured by a neural network model. Then, we use a hierarchical convolutional network to aggregate the information into a hidden vector. Finally, we use the vector to generate the outputs of Actor and Critic.

Grid-Based Representation Model. It's hard for a model to learn the features of spatial distribution from raw input $\mathcal{O}_t$, *i.e.,* coordinates of rectangle data of variable $O(n)$ size. Therefore, we propose to use a grid-based model for representing each state, of which the spatial object distribution is importance for partitioning decisions. Specifically, we divide the universal region of an object set into $W \times W$ equal-size grids and use the statics of rectangles intersecting each grid to represent a state.

If each object is a point, which is either contained by a grid or not, we can simply use 1 integer for each grid to represent the number of points contained by it. However, the possible cases of a rectangle intersecting a grid is not only 2. Therefore we need to use more channels to represent the intersection of a grid with the rectangle data. Specifically, the intersection of 2 rectangles is equivalent to the intersection of their coordinate ranges on each dimension. As Fig. 4a shows, the possible situations in how 2 coordinate ranges intersect is 4, thus there are 4^d cases for a rectangle data to intersect a grid in a d-dimensional space. Therefore, when $d = 2$, we use a 16-dimensional vector for each grid, concatenating 1 extra dimension representing P_t. As a result, as Fig. 4b shows, we can represent s_t by a $W \times W \times 17$ tensor $\mathbf{s}_t$, which is similar to an image ($H \times W \times 3$ with 3 channels of RGB).

Hierarchical Convolutional Network. Next, we aggregate the information from all the grids of $\mathbf{s}_t$ into a hidden vector. We leverage the convolutional neural network (CNN [11]) from the image processing field, which utilizes local perception to effectively capture the spatial features and is usually a basic unit to construct deep network structures such as AlexNet [10].

The network structure is shown in Fig. 5. We use $L = \log_2 W$ layers of CNN and Pooling modules to progressively reduce the $W \times W \times C$ input feature matrix into a $1 \times 1 \times d^h$ hidden embedding $\mathbf{Z}$, aggregating the spatial information from each location. Specifically, in each step i from L down to 1, we use a CNN $Conv_i^{3,3}$ with 3×3 receptive field to extract spatially local correlation from the feature map, followed by a Rectified Linear Unit ($ReLU(x) = max(x, 0)$) and a MaxPooling layer (reducing each 2×2 grids into 1 of the maximum value).

$$\mathbf{Z}_{i-1} = MaxPool^{2,2}(ReLU(Conv_i^{3,3}(\mathbf{Z}_i))), i = L, ..., 1 \tag{2}$$

where $\mathbf{Z}_L = \mathbf{s}_t, \mathbf{Z}_i \in \mathbb{R}^{2^i \times 2^i \times d_i}$ and d_i increases as the matrix size shrinks since it needs to embed more information. Note that $\mathbf{Z} = \mathbf{Z}_0 \in \mathbb{R}^{d^h}$ embeds the global information, extracted by the local perception level by level, which can be used

to generate outputs of Actor and Critic. Also note that the size of the parameter set is small since it does not depend on the number of grids $W \times W$.

Critic Output. The Critic needs to predict the state value (*a.k.a.* return) as $\hat{V}^{\pi}(s_t) \in \mathbb{R}_0^+$. We pass the embedding $\mathbf{Z}$ through a fully connected layer (FC) followed by a ReLU to generate $\hat{V}^{\pi}(s_t)$.

$$\hat{V}^{\pi}(s_t) = ReLU(FC_1(\mathbf{Z})) \in \mathbb{R}_0^+ \tag{3}$$

Actor Output. The Actor network needs to generate an action a_t. We first generate the probability p_{pack} of whether to *Pack*, by an FC layer and the Sigmoid function σ, where $\sigma(x) = \frac{1}{1+e^{-x}} \in (0, 1)$.

$$z_{pack} = FC_2(\mathbf{Z}) \in \mathbb{R}^1, p_{pack} = \sigma(z_{pack}/\tau) \in \mathbb{R} \tag{4}$$

where τ is a temperature parameter controlling the trade-off between exploration and exploitation.

Similarly, we use another FC and the Softmax function (a high dimensional extension of Sigmoid, also denoted as σ) to generate the probabilities $\mathbf{p}_{dim}$ of each of the d dimensions to be chosen as the cutting dimension (Fig. 5 shows a 2-dimensional case that $\mathbf{p}_{dim} = \{p_x, p_y\}$).

$$\mathbf{z}_{dim} = FC_3(\mathbf{Z}) \in \mathbb{R}^d, \mathbf{p}_{dim} = \sigma(\mathbf{z}_{dim}/\tau) \in \mathbb{R}^d \tag{5}$$

Finally, we generate the cut position *pos* and the ratio α of each dimension for when it is chosen as the cut dimension. Since *pos* and α are both continuous values, we use Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ to model them. In the 2-dimensional case (*i.e.,* X and Y), $\mu_x^{pos}, \sigma_x^{pos}$ are generated by:

$$\mathbf{z}_x^{pos} = FC_x^{pos}(\mathbf{Z}) \in \mathbb{R}^2, \mu_x^{pos}, \sigma_x^{pos} = Sigmoid(\mathbf{z}_x^{pos}) \tag{6}$$

Similarly, we can use another 3 FC layers to generate the parameters of Gaussian distributions for Y-axis and for α.

For each step t in the R-tree construction process (Algorithm 2), if the object set can fit in one child node (line 1), we first generate p_{pack} from s_t and sample from the Bernoulli distribution $Bern(p_{pack})$ the decision whether to pack. If not (line 3), we choose a cut dimension from a categorical distribution of $\mathbf{p}_{dim}$, with the cut position *pos* and the node assigning ratio α sampled from the associating Gaussian distributions of that dimension.

5 Model Training

For building R-trees of H levels, we need to train $H - 1$ models. A naive way is repeatedly building R-trees by these models, using the experiences (states, actions, and rewards) to update the Critic model by minimizing Mean squared error (MSE) loss and update the Actor model through PPO [15], a policy gradient method. However, each time we build an R-tree, the experiences provide

unbalanced training samples (one for each decision step) for models of different levels. For example, the model in level H uses $O(B)$ (*e.g.,* several dozens) steps to partition the root node, while the model in level 2 takes $O(B^{H-1})$ (*e.g.,* dozens of thousands) steps to pack all the leaf nodes. Thus it's time-consuming to collect enough training samples for upper levels.

To accelerate this process, we propose to train the models level by level from bottom to up, during which: (1) We can **reuse the parameters** of lower-level models as a relatively good initialization of an upper-level model, and then fine-tune it with fewer training rounds (Sect. 5.1). (2) In each level, we do not build a whole subtree to get all the rewards. Instead, we only partition the current node and use a **short-cut** method to access the rest rewards, *i.e.,* not actually partitioning the child nodes but estimating their rewards by the trained Critic models (Sect. 5.2).

5.1 Bottom-Up Training Sharing

Intuitively, the tasks, *i.e.,* partitioning objects into $x \leq B$ subsets, for Actor-Critic models in various levels are related but different in some details (*e.g.,* child node capacity B^{h-1} and long-term subtree costs). Thus we can reuse a trained model in lower levels as a relatively good initialization of one in the next level.

As shown in Algorithm 3, we train the leaf-level (*i.e.,* $h = 2$) model at first. The training includes repeated exploration and updating rounds (line 3). In each round, we first randomly take an object set $\mathcal{O}$ with $|\mathcal{O}| \in (B^1, B^2]$ and perform *NodePartition* on it, on the policy represented by the current Actor θ_2 (line 4). During this process we collect experiences of all these steps. Then we calculate the long-term costs $V_\pi(s_t)$ (line 5), as a label of the Critic model and an evaluation of the action generated by the Actor model. We use the experiences and all the $V_\pi(s_t)$ to update these models by PPO [15] (line 6). After we finish training the model in level 2, we use its parameters as an initialization of the model in level 3 (line 7), which can be fine-tuned in fewer rounds to converge. Proceeding upwards until the root level, we can have all $H - 1$ models trained.

The calculation of long-term rewards costs most of the time in this process.

5.2 Shortcut Long-Term Reward Calculation

According to Eq. 1, the long-term reward $V_\pi(s_t)$ of a state s_t can be calculated as following: (1) In leaf level, we can simply cumulate the rewards of all the succeeding steps from the collected experiences. *e.g.,* in Level 1 of Fig. 3, $V_\pi(s_4) = r_4 + \gamma_1(r_5 + r_6)$. (2) In non-leaf levels, we need to further partition the child nodes to calculate $V_\pi(s_{t'})$. *e.g.,* in Level 2 of Fig. 3, to calculate $V_\pi(s_1) = r_1 + \gamma_2\{r_3 + \gamma_1[r_4 + \gamma_1(r_5 + r_6) + r_7]\}$, we need to build the whole subtree from s_3. This process is slow, especially for levels near to root, and we propose a short-cut method to avoid further partitions and directly approximate the long-term rewards.

Intuitively, since we have trained the Critic models of lower levels, we can use them to approximate the long-term reward by $V_\theta(s_{t'})$ for each child node $\mathcal{O}_{t'}$ in

Algorithm 3: Bottom-Up Model Training

```
Random initialize Actor-Critic parameters θ_2
for h = 2, ..., H do
    for k = 0, 1, ... do
        Choose an objects set O where |O| ∈ (B^{h-1}, B^h], perform NodePartition
          on policy π(θ_h), get experiences {s_t, r_t, a_t, π(a_t|s_t)}
        Calculate V_t = V_π(s_t) for each t according to Equation 1
        UpdatePPO(θ_h, {s_t, V_t, a_t, π(a_t|s_t)})
    Copy parameters θ_{h+1} = θ_h
```

$O(1)$ time. We use these estimation results as a shortcut for calculating $V_\pi(s_t)$. For example, in Level 2 of Fig. 3, to calculate $V_\pi(s_1) = r_1 + \gamma_2 V_\pi(s_3)$, we use the trained Critic model θ_2 to approximate a $V_{\theta_2}(s_3)$, instead of further partitioning O_3. In this way, we can significantly accelerate the process of collecting training samples while the training quality does not decay too much, since the estimated long-term costs are relatively accurate.

6 Experiments

In this section, we conduct extensive experiments on real-world datasets to evaluate our method **ACR** and compare with the state-of-the-arts including **R*** [2], **HR** [8], **H4R** [6], **STR** [12], **TGS** [4] and **PR** [1].

6.1 Experiment Setup

Implementation Details. We use the codes[2] in [1] including the implementations of **R***, **HR**, **H4R**, **TGS**, **PR** and we also implement **STR**. The proposed R-tree building models are trained on NVIDIA GeForce RTX 3090 GPU, with PyTorch 1.8. The query tests of the R-tree indices are implemented in C++, on 3.10 GHz Intel(R) Xeon(R) Gold 6242R CPU, with 256 GB RAM.

Dataset	Region	#Objects
SD	South Dakota	53,771
WY	Wyoming	100,683
UT	Utah	323,636
AZ	Arizona	1,464,257
TX	Texas	4,081,258
CAL	California	6,321,254

Fig. 6. Datasets

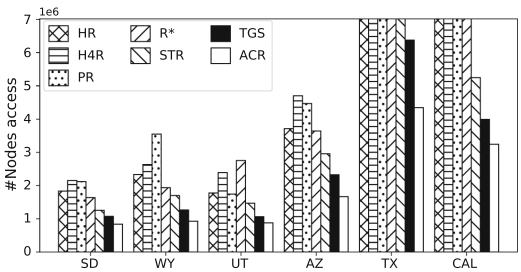

Fig. 7. Range query performance on various datasets

[2] https://www.cse.ust.hk/~yike/prtree/.

Datasets. We use several real-world datasets from OSM[3] including the buildings, represented by rectangles with 2-dimensional coordinates (*i.e.,* longitudes and latitudes), in various regions, as Fig. 6 shows.

R-Tree Parameters. In all the R-trees, we use 40 bytes for each entry in a node, *i.e.,* 32 B for 4 coordinates of an MBR and 8 B for a pointer referencing the corresponding child node (or the ID of an object data in a leaf node). The block size is 4 KB, thus the maximum entry number B of a node is 102. For the grid-based representation (Sect. 4.3), we observe $W = 64$ has the best performance, so we take it as the configuration through all the experiments.

Evaluation. We mainly focus on the number of node access, which is strongly correlated to the I/O costs and processing time, for answering queries. For window query, we generate rectangular windows at random locations, with the height and width randomly sampled from uniform distribution $U(0, 2L)$ with various scale L, and report the objects that intersect each query window. For kNN query, we generate the query points at random locations.

6.2 Results

Window Query Processing. We generate window queries on various datasets where $L = 0.04°$ (around 400 m in ground-distance). The number of queries on each dataset is the region area dividing the query window area. In this setting, the expected sum of the query result (*i.e.,* accessed object data) sizes is equal to the size of each dataset. Figure 7 shows the node access numbers of R-tree built by different methods on various datasets. We make the following observations:

(1) **TGS** and **STR** perform better than the other traditional methods. This is because (a) Sort-based methods **HR** and **H4R** can not well preserve the spatial proximity into 1-dimensional order; (b) **R*** builds an R-tree through one-by-one (*i.e.,* online) insertion, which omits the global structure, thus performs worse than the batch (*i.e.,* offline) building algorithms; (c) **PR** optimizes the worst-case performance and thus may not be the best on most datasets. The node accesses of **TGS** are less than **STR** on all the datasets, because **TGS** greedily optimizes the areas of nodes that are closer to the root, which has a larger impact on query performance.
(2) **ACR** has the best performance on all the datasets, whose node accesses are around 20% to 30% less than those of **TGS**. This is because the DRL-based method (a) considers the long-term tree costs and thus performs better than the greedy strategy, and (b) removes the full-packing constraint that limits the R-tree structure and uses Actor-Critic model to effectively explore a good result.

Varying Window Size and k. We evaluate the performance for range and kNN queries on dataset AZ. The range queries are generated in the same way

[3] http://download.geofabrik.de/.

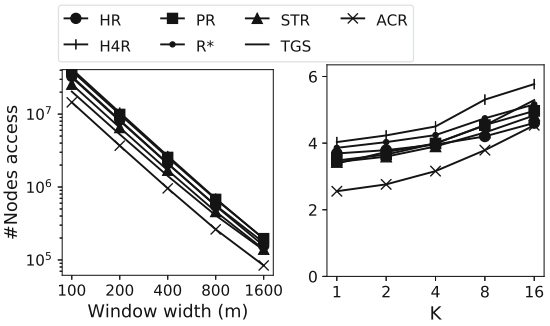

Fig. 8. Window queries of various scales

Fig. 9. kNN queries of various k

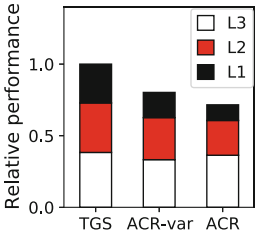

Fig. 10. Performance under various settings

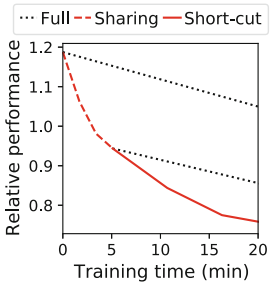

Fig. 11. Effect of bottom-up training

as above, with the window size L varying from 100 m to 1.6 km. The k of kNN queries varies from 1 to 16.

The results are shown in Fig. 8 and 9 and we make the following observations: (1) On all the query sets, the performance of **ACR** is significantly better than all other methods. (2) The node accesses of all the methods decrease as the query window size increases, because the number of queries decreases while the total number of reported objects fixes, thus the total node accesses becomes fewer. (3) For kNN queries, the node accesses increase when k increases, because we need to traverse more nodes to find more nearby objects.

Index Size and Building Time. As Table 1 shows, the index size of **ACR** is slightly larger than other methods. This is because the index size is proportional to the node number. Traditional full-packing methods use the fewest nodes to index all the objects while **ACR** allows nodes to not be full-filled and thus use more nodes. **ACR** also spends more time to build an R-tree because (a) it is implemented in Python which is much slower than C and (b) it passes the state into a neural network to make the decision in each step. However, the time cost is acceptable even for the largest dataset (*i.e.,* CAL of over 6M objects) and the space overhead is small thus **ACR** is practical for usage.

6.3 Self Studies

Effect of Removing Full-Packing Constraint and Long-Term Optimization. To evaluate the effect of overcoming the 2 limitations of **TGS** as introduced in Sect. 3, we train another model **ACR-Greedy** that builds R-trees without the full-packing constraint but only optimizes the area sum of the child nodes but not all the descendants (*i.e.,* the whole subtree), by setting the level discount factor $\gamma_2 = 0$ (in Eq. 1). And we show the numbers of access of nodes relative to TGS in various tree levels which depends on the node areas.

We can observe from Fig. 10 that: (1) **ACR-Greedy** has the better performance than **TGS**. This shows the effect of removing the full-packing constraint, which provides much more space for finding a better tree structure. (2) The node access in the top level (Level 3) has **ACR** more than **ACR-Greedy** because **ACR-Greedy** greedily optimizes the node partitioning of the current level.

Table 1. Index size and building time

Dataset	Index size (MB)		Building time (s)						
	Others	**ACR**	**R***	**HR**	**H4R**	**STR**	**TGS**	**PR**	**ACR**
SD	2	3	0.1	<0.1	<0.1	<0.1	<0.1	<0.1	21
WY	4	5	0.2	<0.1	<0.1	<0.1	0.1	0.1	85
UT	13	13	0.6	0.3	0.4	0.4	0.6	0.7	102
AZ	57	60	3	1	1	1	3	4	447
TX	158	168	7	3	3	3	11	13	1,075
CAL	244	254	12	5	5	5	18	21	1,601

However, **ACR** has much less node access in the rest 2 levels and the total performance is better than **ACR-Greedy**. This is because **ACR** optimizes the long-term tree costs, which can build an overall better R-tree.

Effect of Bottom-Up Training. To evaluate the effect of the bottom-up training framework along with the training sharing and the short-cut strategies, we train models of 2 levels on WY in different ways and report the query performance relative to TGS in Fig. 11. If we directly build the whole tree to train the 2 models (**Full**), the time cost is high. If we first train the model of level 1, of which the time cost is low, in the meanwhile sharing its parameters with the level 2 model (**Sharing**), it achieves good performance in much less time. Then we partition the tree nodes in level 2 and train the associating model. If we get the long-term rewards $V_\pi(s_t)$ by further partitioning the child nodes (**Full**), the speed is slow, while using the short-cut method to estimate $V_\pi(s_t)$ (**Short-cut**) costs much less time.

7 Conclusion

In this paper, we propose ACR-tree a DRL-based R-tree building algorithm, which overcomes 2 limitations of the existing methods. First, we propose tree-MDP to model the long-term tree costs, which is omitted by the traditional methods. Second, we remove the full-packing constraint and use Actor-Critic models to effectively explore the resulting huge search space, with a hierarchical CNN structure for embedding the spatial distribution of objects to effectively estimate the long-term costs and make the node partitioning decisions. We also propose a bottom-up framework with 2 strategies, *i.e.,* training-sharing and shortcut, to efficiently train the models. Extensive experiments on real-world datasets show that the ACR-tree significantly outperforms existing R-trees.

Acknowledgement. This paper was supported by National Natural Science Foundation of China (61925205, 62232009), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology.

References

1. Arge, L., Berg, M.D., Haverkort, H., Yi, K.: The priority R-tree: a practically efficient and worst-case optimal R-tree. ACM Trans. Algorithms (TALG) **4**(1), 1–30 (2008)
2. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pp. 322–331 (1990)
3. Beckmann, N., Seeger, B.: A revised R*-tree in comparison with related index structures. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 799–812 (2009)
4. García R, Y.J., López, M.A., Leutenegger, S.T.: A greedy algorithm for bulk loading R-trees. In: Proceedings of the 6th ACM International Symposium on Advances in geoGraphic Information Systems, pp. 163–164 (1998)
5. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp. 47–57 (1984)
6. Haverkort, H., Walderveen, F.V.: Four-dimensional Hilbert curves for R-trees. J. Exp. Algorithmics (JEA) **16**, 3-1 (2008)
7. Kamel, I., Faloutsos, C.: Hilbert R-tree: an improved R-tree using fractals. Technical report (1993)
8. Kamel, I., Faloutsos, C.: On packing R-trees. In: Proceedings of the Second International Conference on Information and Knowledge Management, pp. 490–499 (1993)
9. Konda, V., Tsitsiklis, J.: Actor-critic algorithms. Adv. Neural Inf. Process. Syst. **12** (1999)
10. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Adv. Neural. Inf. Process. Syst. **25**, 1097–1105 (2012)
11. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
12. Leutenegger, S.T., Lopez, M.A., Edgington, J.: STR: a simple and efficient algorithm for R-tree packing. In: Proceedings 13th International Conference on Data Engineering, pp. 497–506. IEEE (1997)
13. Li, G., Zhou, X., Li, S., Gao, B.: QTune: a query-aware database tuning system with deep reinforcement learning. Proc. VLDB Endow. **12**(12), 2118–2130 (2019)
14. Qi, J., Tao, Y., Chang, Y., Zhang, R.: Theoretically optimal and empirically efficient R-trees with strong parallelizability. Proc. VLDB Endow. **11**(5), 621–634 (2018)
15. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
16. Sellis, T., Roussopoulos, N., Faloutsos, C.: The R+-tree: a dynamic index for multi-dimensional objects. Technical report (1987)
17. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
18. Yu, X., Li, G., Chai, C., Tang, N.: Reinforcement learning with tree-LSTM for join order selection. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 1297–1308. IEEE (2020)