Course: Complex Data Management

Project 2: Geospatial Data

Name: Konstantinos Vardakas

Student ID: 522

Email: pcs0522@uoi.gr

Part 1: R-Tree construction via bulk loading

The first part of the project involves the construction of an R-Tree using the bulk loading method. The relevant code can be found in the files rtree_init_functions.py (helper functions) and rtree_init_main.py (main program). The process begins by reading the files containing the data coords.txt and offsets.txt, using the function read_data(coords_path, offsets_path), which returns two lists. The first list returned, the coords list, contains the coordinates of the point (coords = [[x1, y1], [x2, y2], …]) and the second contains information about the grouping of points to create spatial data (offsets = [[id1, start_ind1, end_ind1], [id2, start_ind2, end_ind2], …]).

Next, the MBR window that includes each polygon must be calculated by finding the minimum and maximum values of each axis (x and y) for each polygon (function compute_mbr(coords)). Once the MBR has been determined, its center is found by calculating the average of the minimum and maximum values for each axis (function mbr_center(mbr)). This center is then used to extract the z-order value of each MBR, which helps group the polygons when constructing the R-Tree. This calculation of the Z-order value for a given MBR center is done using the interleave_latlng(y, x) function from the pymorton library. The build_mbrs(coords, offsets) function uses the previous functions to create the data, polygons, and z values, and the returned list contains the data sorted by z-order value. Because the records in the leaves created next must be of the type [id, MBR], the z-order value is deleted from the returned list.

In order to have the data ready in leaves, the group_entries(entries) function was created, which groups the data into twenties, which is the maximum number that each leaf can have. If there is excess data above the minimum allowable limit (min_entries = 8), another group is created. However, if the excess is less than this limit, the last

complete group entered is removed and recreated after moving enough data from it to fill another group with 8 (or min_entries) data.

At this point, the data is ready in leaves, and the nodes can be created. This process is performed with the build_nodes(entries, node_type) function. The grouping into leaves is performed within the function, and then for each leaf, the MBR is calculated, which includes all the MBRs of the data in a leaf (using the function compute_union_mbr(mbr_list)). Thus, for a given leaf, a node is created with MBR, with children the leaf itself, and with type = 'leaf', which will then distinguish the leaf nodes from the rest. Next, a node id is assigned, starting with the numbering of the first leaf (leaf with the data with the smallest z-order curve). The same function is used to create the parent nodes, as each node created also has an MBR.

Now the tree can be created using the build_rtree function, which first creates the leaf nodes as described above and then assigns IDs to each node. Next, the subsequent levels of the tree (node_type='node') are built until the level being built has between 2 and 20 nodes, which are grouped under the root node.

To print the nodes per level, a queue containing tuples of the form (node, level) was used, which was initialized with the root node at level 0. The numbering of the levels was used exclusively to monitor the structure during processing, as the results are ultimately printed in reverse order, starting from the leaves and ending at the root. More specifically, the process involved removing the first element from the queue (node, level), recording the level at which the node is located, and then inserting all its children into the queue as (child_node, level + 1) for further processing. When the process reaches the leaves of the tree, which have no further children, no new elements are added to the queue. Thus, the queue is gradually emptied and the counting is completed. Once the level_counts dictionary containing the collected information has been completed, the order of the levels is reversed so that printing starts from the leaves and ends at the root, displaying the number of nodes at each level.

To store the R-Tree structure in a .txt file, the write_rtree_to_file_dfs_sorted function was implemented. The process begins with a recursive depth-first search (DFS) of the tree, starting from the root. During this search, each node of the tree is recorded in a list of entries, which contains lists of the form [isnonleaf, id, children_data]. The isnonleaf field indicates whether the node is internal or a leaf, while

children_data includes the ids and MBRs of the node's children. After the search is complete, the list of entries is sorted based on the id of each node. Finally, the sorted data is stored in the Rtree.txt file, with each line corresponding to a node in the tree.

Part 2: Range Queries

The first part of this section is loading the rtree from the previously saved file. This is done with the load_tree function, which recreates the R-Tree from the text file that was generated when the tree was saved in the first part of the task. Initially, the function reads the contents of the file and, using ast.literal_eval, parses each record into the three elements that were previously stored during the tree recording process: an integer index (isnonleaf) indicating whether the node is internal or a leaf, the unique node identifier (node_id), and the list of its children in the form of pairs [child_id, mbr]. With this data, a nodes dictionary is created, where each entry contains basic information about the node, such as its type (leaf or internal node), the list of its children (initially in the form [id, mbr]), and the identifier itself.

Next, the child_ids in the child lists are replaced with the corresponding full dictionaries of the child nodes. This replacement is not done with deepcopy, which preserves common references between parent and child nodes, ensuring data consistency. The leaves of the tree maintain a simpler structure, containing only id and mbr. In the last stage, the node with the maximum id is located, which is considered the root, its mbr is calculated by merging the mbrs of its children (since it was not stored during registration), and it is modified to have the original tree structure. The image below shows that the reconstructed tree is identical to the original, as the storage and reading of the nodes retain the same structure and content. Despite the slight difference in the order of the root node fields (where in the original tree the mbr appears last, while in the reconstructed tree it precedes the id), the nodes are equivalent and produce the same hash values, confirming that the representation of the tree has not been affected.

```
coords, offsets = read_data('coords.txt', 'offsets.txt')
objects = build_mbrs(coords, offsets)
objects = delete_z(objects)
rtree_root = build_rtree(objects)
write_rtree_to_file_dfs_sorted(rtree_root)
rtree = load_tree()

with open('Rtree.txt', 'r') as f:
    initial = f.read()

with open('Rtree_temp.txt', 'r') as f:
    second = f.read()

hash(initial) == hash(second)
```

```
True
```

Figure 1: Comparison of the hash values of the two files that have the tree nodes with a common hash function

Next, to implement the range query, R-tree Depth First Search was used, starting from the root and examining at each step whether the minimum MBR of a node intersects with the query window w. If there is no intersection, the subtree below that node is ignored. The implementation is based on the mbrs_intersect(w, mbr) function, which geometrically checks whether two MBRs intersect in two dimensions. When the recursion reaches leaves, each object is checked separately, and those that intersect with the query window are added to the results list.

Part 3: k-NN queries

To implement the kNN query, the Best-First Search technique was used, with priority given to the proximity of each node or data point to the query point q. The search starts from the root of the tree and maintains a min-heap (priority queue) where candidate nodes or leaves are stored based on their minimum Euclidean distance from point q. The point_to_mbr_distance function calculates the distance of a point from an MBR, returning 0 if the point is inside the MBR, or otherwise the shortest distance from the boundary.

Each time the closest element is removed from the queue, if it is a leaf, it is added to the list of results. If it is an internal node, its children are added to the queue with their distance from the point calculated. The process is repeated until the k nearest objects are found. A counter is also used as a tie-breaker to avoid comparisons between

objects that cannot be classified directly (e.g., dictionaries). This approach ensures that the search is always directed towards areas of space that are most likely to contain nearby objects, significantly improving efficiency.

Instructions for execution

To execute each file, the following reasoning is followed:

- Open the terminal in the .py files folder.
- Add the data files to the same folder.
- The general command is: python {scriptname.py} args

For each file separately:

- python rtree_init_main.py [coords_path] [offsets_path] [output_path] (default values 'coords.txt', 'offsets.txt', 'Rtree.txt' respectively)
- python range_queries_main.py [rtree_file] [queries_file] (default values 'Rtree.txt', 'Rqueries.txt')
- python knn_queries_main.py [rtree_file] [queries_file] [k] (default values 'Rtree.txt', 'NNqueries.txt', 10)

The filename can be given either with or without the .txt extension.