COMP7107 Assignment2 README

3035021742 Zhang Zhuan

General Info:

- The programs are consisted of three parts: Part1 processing data of multiple polygons and building a R-tree structure as database index; Part2 checking range queries basing on the R-tree structure in Part1 and window data; Part3 carrying out the k Nearest Neighbors queries basing on the R-tree structure and points data.
- Written in Jupyter Notebook/ Python 3.6.
- To run the data please prepare data files including: a coordination dataset, an offsets dataset, a range queries dataset and a NN queries dataset.
- Python Library applied: NumPy, pymorton, math and ast.

Part1 Building R-tree (only demonstrate keep steps)

1. Read the coordination.txt and offsets.txt file. Loop in the offsets file to link the point coordinations into polygons, then calculate the MBR of the polygons.

```python
# combine coords and offsets to read the polygons
# calculate MBR and global MBR of the polygons
polygonsMBR = []

def MBR(coords):
    min_x, min_y = np.min(coords, axis=0)
    max_x, max_y = np.max(coords, axis=0)
    return [min_x, max_x, min_y, max_y]
```

```python
for i in offsets:
    poly_id = i[0]
    startOffset = i[1]
    endOffset = i[2]+1
    poly = coordinates[startOffset:endOffset]
    theMBR = MBR(poly)
    polygons = [poly_id, theMBR]
    polygonsMBR.append(polygons)
```

2. Calculate the z-order of the geometric centers of the MBRs to sort them into a list, in which closer items in list are also closer in 2D space.

```python
# calculate geometric centers
# compute the z-order value of the centers (interleave_latlng)
polygonsMBRunsort = []

for i in polygonsMBR:
    ave_x = (i[1][0]+i[1][1])/2
    ave_y = (i[1][2]+i[1][3])/2
    Geocenter_zscore = pm.interleave_latlng(ave_y,ave_x)
    polygons = [i[0],i[1],Geocenter_zscore]
    polygonsMBRunsort.append(polygons)

# sort the z-order value of the MBR
polygonsMBRsorted = sorted(polygonsMBRunsort, key=lambda x:x[2]) #with z

MBRsorted = [] #without zscore
for i in polygonsMBRsorted:
    polygons = i[:2]
    MBRsorted.append(polygons)
```

3. Before I build the R-tree. I can calculate the total number of leaf nodes of the R-tree, the number of total levels of the tree as well as the number of non-leaf nodes of each level.
Using the node numbers of each level as list slicing index, I can easily form a well structure R-tree from the polygons' MBR list.

```python
# set the node capacity for the Rtree
nodecapacityM = 20
nodecapacitym = 0.4*nodecapacityM

#calculate the numbers of leaf node in the Rtree
leafnodeNo = math.ceil(len(MBRsorted)/nodecapacityM)

#calculate the numbers and levels of nonleaf node part of the Rtree
nonleafnodeNo = math.ceil(leafnodeNo/nodecapacityM)
nonleafnode_level = [nonleafnodeNo]
while nonleafnodeNo > 1:
    nonleafnodeNo = math.ceil(nonleafnodeNo/nodecapacityM)
    nonleafnode_level.append(nonleafnodeNo)

Rtreelevel = len(nonleafnode_level)+1
```

4. Build the R-tree:

4.1 Form the list of all leaf nodes of the Rtree

- to make sure that the last node of the leaf level contains data over 40% of the capacity M;

- append the leaf nodes into the list in the format of [0, id of the leaf node, [id of the polygons, polygons' MBR]]

```
#construct the leaf nodes of the Rtree
#assume that each R-tree node has maximum capacity 20 and minimum number
Rtree_leaf = []
idNo = 0

if 0 < len(MBRsorted)%nodecapacityM < nodecapacitym:
    MBRsorted_cut = MBRsorted[:int(-nodecapacitym)]
    MBRsorted_tail = [MBRsorted[int(-nodecapacitym):]]
    nodelist0 = [MBRsorted_cut[i:i+20] for i in range (0, len(MBRsorted_
    nodelist0.extend(MBRsorted_tail)

else:
    nodelist0 = [MBRsorted[i:i+20] for i in range (0, len(MBRsorted), no


for i in nodelist0:
    leafnode = [0, idNo, i]
    Rtree_leaf.append(leafnode)
    idNo += 1
```

4.2 Construct the upper level of the tree:

- define a function that can cut a list of nodes with MBR inside into non-leaf nodes, then calculate the global MBR of the non-leaf nodes basing on their attached nodes;

with the function, the program can process R-tree structures that have difference levels comparing to the given dataset;

-the id of the non-leaf nodes is not repeated from the leaf nodes ;

- build a R-tree list with 4 items corresponding to 4 levels;

- or take off the level division the R-tree;

```
# construct the upper levels of the tree
def build_rtree(lst):
    theMBR = []
    Rtree_nonleaf0 = []
    global idNo
    nodeid = 0
    for i in lst:
        nodeMBR = []
        for j in i[2]:
            nodeMBR.append(j[1])
        globalnodeMBR = [nodeid, global_MBR(nodeMBR)]
        theMBR.append(globalnodeMBR)
        nodeid += 1

    if 0 < len(theMBR)%nodecapacityM < nodecapacitym:
        MBRsorted_cut = theMBR[:int(-nodecapacitym)]
        MBRsorted_tail = [theMBR[int(-nodecapacitym):]]
        nodelist1 = [MBRsorted_cut[i:i+20] for i in range (0, len(MBRsor
        nodelist1.extend(MBRsorted_tail)
    else:
        nodelist1 = [theMBR[i:i+20] for i in range (0, len(theMBR), node

    for i in nodelist1:
        nonleafnode = [1, idNo, i]
        Rtree_nonleaf0.append(nonleafnode)
        idNo += 1
    return Rtree_nonleaf0


idNo = len(Rtree_leaf)
Rtree = [Rtree_leaf]

#build a Rtree with level division
for i in range(Rtreelevel-1):
    Rtree_nonleaf = build_rtree(Rtree[-1])
    Rtree.append(Rtree_nonleaf)

#build a Rtree without level division
Rtree_withoutlevel = []
for i in Rtree:
    for j in i:
        Rtree_withoutlevel.append(j)
```

Part2 Range Queries (only demonstrate keep steps)

1. Read the Rtree.txt file written in Part1. And transfer the string file into a list with all the nodes of the R-tree.
The nodes are not splitted into different levels yet.

```
StrRtree = []
with open(Rtreefile,'r') as file:
    for line in file:
        StrRtree.append(line)

# convert string of the input Rtree file to list and keep the list hiera
def convert_list(lst):
    newlist = []
    for i in lst:
        if isinstance(i, list):
            newlist.append(convert_list(i))
        else:
            newlist.append(i)
    return newlist

Rtree = []
for string in StrRtree:
    singlelist = ast.literal_eval(string)
    newlist = convert_list(singlelist)
    Rtree.append(newlist)
```

2.1 By counting the nodes with 0 in the first place, I can have the number of leaf nodes. With the similar calculation logic in Part1, the total number of Rtree levels and numbers of nodes in each level are reached.

```
# recount the nodes numbers and how many levels for the Rtree
# set the node capacity for the Rtree
nodecapacityM = 20
nodecapacitym = 0.4*nodecapacityM

#calculate the numbers of leaf node in the Rtree
leafnodeNo = sum(1 for nodes in Rtree if nodes[0] == 0)

#calculate the numbers and levels of nonleaf node part of the Rtree
nonleafnodeNo = math.ceil(leafnodeNo/nodecapacityM)
nonleafnode_level = [nonleafnodeNo]
while nonleafnodeNo > 1:
    nonleafnodeNo = math.ceil(nonleafnodeNo/nodecapacityM)
    nonleafnode_level.append(nonleafnodeNo)

Rtreelevel = len(nonleafnode_level)+1
NLN_level = nonleafnode_level[::-1]
```

2.2 With the numbers above, I can split the list of all nodes without hierarchy into 4 levels Rtree list.

```
#rebuild Rtree
NewRtree = []

for i in NLN_level:
    nonleafnode_list = Rtree[-i:]
    del Rtree[-i:]
    NewRtree.append(nonleafnode_list)

NewRtree.append(Rtree)
```

3. Define the range query function that can return the id of the nodes or polygons which intersect with the query windows.

```
# range query function
def WQ_xycompare(j):
    xmin = j[1][0]
    xmax = j[1][1]
    ymin = j[1][2]
    ymax = j[1][3]

    if xmax< Wx_min or xmin > Wx_max or ymax < Wy_min or ymin > Wy_max:
        return
    else:
        return j[0]

def delNone(lst):
    newlist = [x for x in lst if x is not None]
    return newlist
```

4. Define the Rtree scan function that can return the intersected nodes ID of the first item in the Rtree list, then delete the scanned level from the list.

```python
# Scan Rtree function, iteratly build index of each level's intersection
def scan_rtree(lst):
    global index

    if len(lst) != 0:
        newindex = []
        for i in index:
            for j in lst[0][i][2]:
                newindex.append(WQ_xycompare(j))
        newindex = delNone(newindex)

    del lst[0]
    return newindex
```

5. Loop in the range queries and keep feed the result of the scan_rtree function to itself so that there is no abortive scan of the irrelavent nodes. In the end, the function will return the intersected polygons's id. Then a range query scan of the Rtree is completed.

```python
# apply the list of range queries to the Rtree and print the result
WQid = 0
for wq in Wqueries:
    Wx_min = wq[0]
    Wx_max = wq[2]
    Wy_min = wq[1]
    Wy_max = wq[3]

    ScanRtree = NewRtree[:]

    index = [0]

    for i in range(len(NewRtree)):
        scan_index = scan_rtree(ScanRtree)
        index = scan_index

    print(str(WQid)+'('+str(len(index))+'): '+','.join(str(x) for x in i
    WQid += 1
```

the first 6 queries' results of Part2

```
0(7): 2527,2712,8371,5042,7080,7656,7944
1(101): 7899,3508,1018,3394,8090,9349,3364,475,3999,764,2738,4606,9362,
9303,6073,6389,3048,6714,5898,4990,3107,1980,5792,8019,1006,6294,7924,7
923,6374,9987,3185,1044,3325,6902,9974,2294,8842,1506,8207,8475,666,891
0,1277,7501,2249,5292,5499,4978,3507,8086,1440,4656,3044,6530,8970,3174
,4727,817,7108,3430,9919,83,8594,6147,6256,5592,2864,2708,8367,1619,200
8,3882,5037,2117,3140,151,2108,5597,7770,5872,3845,4073,7430,6442,420,1
64,532,6571,5916,1600,2359,8747,5304,8415,4734,5030,8865,5562,8335,1051
,6493
2(98): 9039,4158,6710,6611,702,5290,6975,2428,1500,2735,2424,4379,9603,
2385,4587,7593,3627,210,2136,5440,5696,6727,9591,8312,4027,7866,9268,38
23,5111,5842,8771,4685,7759,8197,6999,1484,6921,8036,954,8863,7410,463,
4058,6845,6607,486,125,240,9888,792,1915,9338,7828,4127,2662,1920,3533,
6230,6027,3075,1014,7600,1595,7197,7964,9233,4394,3705,3801,2467,402,20
70,8102,2308,7984,382,3371,7562,1304,926,1088,6765,720,5241,4406,5749,8
283,5114,6900,1342,6198,6783,9641,5470,5523,4880,6151,8323
3(44): 877,6285,4225,8928,1732,8090,3048,3107,7924,7923,6374,1044,3325,
6902,1277,7501,4978,8086,3174,9919,83,8594,6147,2117,2108,532,2359,4734
,5030,8865,5562,8335,2272,3180,1987,4372,1131,7011,3712,2351,6324,5902,
3077,5595
4(56): 3650,1470,4004,2963,5979,9228,7712,2149,6163,8671,1795,5043,181,
5864,5607,1136,3056,7937,8779,6314,8794,73,5901,3317,4732,1364,393,1728
,5572,1422,6664,861,1283,5357,7067,9713,9470,6585,9867,2468,2718,60,750
4,5157,1530,8044,8082,5202,2497,6566,1952,2791,152,1688,3803,2903
5(39): 5110,7761,4704,514,3730,8884,6531,1508,8515,579,4521,5771,2084,8
018,9078,9960,1046,9515,9582,8477,2788,2102,2982,7546,3549,4714,6721,26
96,4919,869,1531,3585,1592,5581,1591,3619,7647,897,1673
```

Part3 k Nearest Neighbors Queries (only demonstrate keep steps)

1. The steps of rebuild R-tree structure list are the same as Part2.
To find the nearest neighbors, I need a function to calculate the distance between the MBR and the given query point.

```python
# calculate euclidean distance of the MBR to the query point
def check_distance(Dq,MBR):
    Xd = Dq[0]
    Yd = Dq[1]

    X1 = MBR[1][0]
    Y1 = MBR[1][2]
    X2 = MBR[1][1]
    Y2 = MBR[1][3]

    Xn = max(X1, min(Xd, X2))
    Yn = max(Y1, min(Yd, Y2))

    Distx = Xn - Xd
    Disty = Yn - Yd
    return np.sqrt(Distx**2 + Disty**2)
```

2. As the processing in the Part2, we need a function to return the id of the qualified nodes and polygons.
In this function, the scan is also limited to the global index list which is the returned result of higher level nodes scan. Then keep the qualified items sorted by distance and keep the total number of the items <= k.

```python
# define the k Nearest Neighbors query by reach the closest node MBRs i
def kNN_rtree(lst,Dq,k):
    global index

    if len(lst) != 0:
        newheap = []
        for i in index:
            for j in lst[0][i][2]:
                distance = check_distance(Dq, j)
                newheap.append((distance,j[0]))
                newheap = sorted(newheap, key=lambda x:x[0])
                if len(newheap)>k:
                    del newheap[-1]
        heapindex = []
        for i in newheap:
            heapindex.append(i[1])

    del lst[0]
    return heapindex
```

3. Loop in the NNqueries file and define the initial index as 0 (as there is only one root node) as well as the k.
In the end, the function will return the k Nearest Neighbors of each queries. Then a kNN query scan of the Rtree is completed.

```python
DQid = 0
for Dq in NNqueries:
    kNNRtree = NewRtree[:]

    index = [0]
    k = 10

    for i in range(len(NewRtree)):
        heap_index = kNN_rtree(kNNRtree, Dq, k)
        index = heap_index

    print(str(DQid)+'('+str(len(index))+'): '+','.join(str(x) for x in i
    DQid += 1
```

the first 10 queries' results of Part3

```
0(10): 9311,7001,803,5361,6764,3905,1642,3260,4669,5762
1(10): 6764,1642,9311,7001,4888,3260,3905,4669,803,5361
2(10): 9311,7001,803,5361,6764,3905,1642,3260,5762,4669
3(10): 6764,1642,9311,7001,4888,3260,3905,4669,803,5887
4(10): 803,9311,5361,7001,6764,4669,1642,6595,9874,3905
5(10): 9311,6764,7001,1642,803,4669,3905,3260,4888,5361
6(10): 6764,1642,9311,7001,4669,4888,3260,3905,803,5361
7(10): 6764,4669,1642,9311,7001,4888,6595,803,9874,5361
8(10): 9311,803,7001,5361,6764,1642,3905,4669,3260,5762
9(10): 6764,1642,9311,7001,4669,4888,3260,3905,803,5361
```