AS data center network approximation algorithm

Data Structures:

<table>
<thead>
<tr>
<th>Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(asn, lat, lng, size) ← block</td>
</tr>
<tr>
<td>(x, y, w, h) ← rectangle</td>
</tr>
<tr>
<td>(rectangle, parentNode, leftChildNode, rightChildNode, setOfBlocks) ← node</td>
</tr>
<tr>
<td>Map(id, setOfBlocks) ← clusters</td>
</tr>
<tr>
<td>Map(block, id) ← labelMap</td>
</tr>
</tbody>
</table>

Helper functions:

- `haversineDistance(lng0, lat0, lng1, lat1)`: calculates the distance between two locations considering the spherical properties of the earth.
- `calculateAvgW(rectangle) / calculateAvgH(rectangle)`: calculates average width / height considering the spherical properties of the earth and the extent of the bounding rectangle, uses Haversine distance.
- `calculateBoundingRectangle(setOfBlocks)`: calculates the bounding rectangle that encompasses all geographical locations of the block set.
- `calculateArea(rectangle)`: calculates the area of the bounding rectangle considering the spherical properties of the earth, uses Haversine distance.
- `getLargestBlock(setOfBlocks)`: returns the block with the biggest size.
- `calculateCenter(setOfBlocks)`: calculates the center position of the block set weighted by the size of the block.
- `getNearestBlock(lat, lng, setOfBlocks)`: returns nearest block from the set to the specified geographical coordinates.
- `rangeQuery(setOfBlocks, block, r)`: returns a set of blocks that are in the radius r of the query block, uses Haversine distance.
- `convertToClusterMap(labelMap)`: converts the map of tuples (block, id) to a map of (id, setOfBlocks) aggregating blocks with the same id to a set of blocks.
- `sortBySize(setOfBlocks)`: sorts a set of blocks ascending by their size.
**Construction of the k-d tree:**
Generate k-d tree by recursively splitting nodes, starting with a node containing all blocks of an ASN and its bounding rectangle. The additional maxDistance parameter steers the stopping criterium of the algorithm.

```plaintext
split(node, maxDistance)
    if node.setOfBlocks.size < 2 then
        return
    avgW ← calculateAvgW(node.rectangle)
    avgH ← calculateAvgH(node.rectangle)
    if avgW < maxDistance ∧ avgH < maxDistance then
        return
    leftData ← {}
    rightData ← {}
    if avgW > avgH then
        splitLng ← node.rectangle.x + node.rectangle.w / 2
        for b in node.setOfBlocks
            if b.lng < splitLng then
                leftData ← leftData ∪ b
            else
                rightData ← rightData ∪ b
    else
        splitLat ← node.rectangle.y + node.rectangle.h / 2
        for b in node.setOfBlocks
            if b.lat < splitLat then
                leftData ← leftData ∪ b
            else
                rightData ← rightData ∪ b
    leftRectangle ← calculateBoundingRectangle(leftData)
    rightRectangle ← calculateBoundingRectangle(rightData)
    (leftRectangle, node, NIL, NIL, leftData) ← leftChild
    (rightRectangle, node, NIL, NIL, rightData) ← rightChild
    node.leftChildNode ← leftChild
    node.rightChildNode ← rightChild
    split(leftChild)
    split(rightChild)
```

**Calculation of clusters and centroids**
After the coarse split of the block data with the k-d tree we traverse the cells and merge neighboring blocks to clusters with a customized DBSCAN algorithm. Centroids of these clusters form the approximated data center locations.

```plaintext
calculateClusters(node, minBlockSize)
    setOfBlocks ← {}
    if node.leftChildNode ≠ NIL then
        setOfBlocks ← setOfBlocks ∪ calculateClusters(node.leftChildNode)
    if node.rightChildNode ≠ NIL then
        setOfBlocks ← setOfBlocks ∪ calculateClusters(node.rightChildNode)
    else
        setOfBlocks ← setOfBlocks ∪ calculateCentroidsDBSCANBased(node.setOfBlocks, node.rectangle, minBlockSize)
    return setOfBlocks
```
Triggers the DBSCAN algorithm and calculates centroids based on the result. Considers area and the number of blocks in the bounding rectangle for the selection of the radius and the minPoints parameters of the DBSCAN algorithm. Additional parameter: minBlockSize, used as a threshold to discard small blocks that were categorized as noise by the DBSCAN algorithm.

```python
def calculateCentroidsDBSCANBased(setOfBlocks, rectangle, minBlockSize):
    if setOfBlocks.size == 1:
        return setOfBlocks

    centroids = {}
    area = calculateArea(rectangle)
    numBlocks = setOfBlocks.size

    r = 1.2 * (area / numBlocks)**0.5
    minPoints = 0.5 * numBlocks**0.5

    clusters = dbscan(setOfBlocks, r, minPoints)

    if clusters.size == 1 and clusters[0].id == -2:
        b = getLargestBlock(setOfBlocks)
        centroids = centroids | b
        return centroids

    for c in clusters:
        if c.id == -2:
            for noise in c.setOfBlocks:
                if noise.size >= minBlockSize:
                    centroids = centroids | noise

        if c.setOfBlocks.size == 0:
            continue

        if c.setOfBlocks.size == 1:
            centroids = centroids | c.setOfBlocks[0]
            continue

        center = calculateCenter(c.setOfBlocks)
        nearest = getNearestBlock(center[0], center[1], c.setOfBlocks)
        centroids = centroids | nearest

    return centroids
```
**DBSCAN**
The resulting clusters are labeled by their id: clusters: 0…n, undefined: -1, noise: -2

```python
dbscan(setOfBlocks, r, minPoints)
    labelMap ← {}
    for p in setOfBlocks
        labelMap.put(p, -1)

    clusterId ← 0
    for p in setOfBlocks
        if labelMap.get(p) ≠ -1 then
            continue

        neighbors ← rangeQuery(setOfBlocks, p, r)
        if neighbors.size < minPoints then
            labelMap.put(p, -2)

        clusterId ← clusterId + 1
        labelMap.put(p, clusterId)

        set ← {}
        for b in neighbors
            if p ≠ b then
                set ← set ∪ b

        index ← 0
        while index < set.size
            q ← set[index]
            index ← index + 1
            if labelMap.get(q) = -2 then
                labelMap.put(q, clusterId)
            if labelMap.get(q) ≠ -1 then
                continue
            labelMap.put(q, clusterId)

        neighbors ← rangeQuery(setOfBlocks, q, r)
        if neighbors.size ≥ minPoints then
            for nBlock in neighbors
                if ¬(nBlock ∈ set) then
                    set ← set ∪ nBlock

    clusters ← convertToClusterMap(labelMap)
return clusters
```
Postprocessing
In a postprocessing step we merge clusters that are located closer to each other than a specified threshold.

```python
mergeClusters(setOfBlocks, distance)
    clusters ← sortBySize(setOfBlocks)
    index ← 0

    while index < clusters.size
        b ← mergedClusters[index]
        mergeSet ← {}

        for i in {index+1 … mergedClusters.size}
            d = haversineDistance(b.lng, b.lat, clusters[i].lng, clusters[i].lat)
            if d < distance then
                mergeSet ← mergeSet U clusters[i]

        if mergeSet.size > 0 then
            blockSize ← b.size
            for mb in mergeSet
                blockSize ← blockSize + mb.size
                mergeSet ← mergeSet \ mb

            newBlock ← (b.asn, b.lng, b.lat, blockSize)
            clusters[index] ← newBlock

            index ← index + 1

        return clusters
```