## Part 4

In this part, we applied a region growing algorithm in order to perform the segmentation on a 3D vascular structure scan. The scanned object is in type of NIfTI file. We have used nibabel library to read from and write to the segmentation data.

For segmentation, we have used region growing algorithm as project suggests. There were 2 hyperparameters; growing criteria and threshold value to grow. There are 4 tasks, 2 of them requires us to apply 2D segmentation to every slice of 3D object by allowing us to choose 15 seed points at most, first one required us to do it by using 8-neighborhood and the second one required us to do it by using 4-neighborhood. The last 2 of the tasks required us to apply 3D segmentation, segmentation to the whole object, the first task of 3D segmentation required us to apply 26-neighborhood, the second task required us to apply same region growing with 6-neighborhood, only allowing to choose 5 seed points at most.

### Algorithm

The region growing segmentation algorithm works like this;

1. We search for the seed point by looking at the brightest point on the image. Since there exist some amount of noise, the points that we need to segment are the brightest ones. If the brightest one has already marked as visited, we search for second brightest point which has not been visited yet.

```
def findSeedPoints(self):
"""
Choose the brightest not visited point
If there is a point that is less than or equal to 0.7,
the function returns an empty list.
"""
    elements = list(np.unique(self.img))
    elements.reverse()
    for element in elements:
        if element <= 0.7:
            break
        x, y, z = np.where(self.img == element)
        if self.visited[x[0], y[0]] != 1:
            return [x[0], y[0]]
    return []
```

2. After finding a seed point to grow from, we start growing using Breadth-first search algorithm. If 8-neighborhood growing is selected, we traverse those all and check whether they are inside the boundaries, whether the difference between intensity of the new point and average intensity of marked points is less than specific threshold, and whether that point that is being visited hasn't been visited before. Satisfying all 3, we add our new point to average intensity calculation, append at the end of the queue, and mark it as visited.

```
def isVisited(self, new_node):
    """
    Check if the new node is visited, if not return true else false.
    """
    return bool(self.visited[new_node[0], new_node[1]])
```

```
 def checkThreshold(self, new_node):
     """
     Calculate mean of positive nodes. If new node is less than threshold,
     return false, else true.
     """
     new_point = self.img[new_node[0], new_node[1]]
     return (np.abs(self.pointMean - new_point) < self.threshold)
 def checkIfOut(self, new_node):
     """
     Check if candidate node is not out of bounds
     """
     if new_node[0] < 0 or new_node[1] < 0  or new_node[0] >= self.height or
new_node[1] >= self.width:
         return False
     return True
```

3. We continue this process, until no brighter points greater than 0.7(the value we have chosen) have left. After finishing growing from the last seed, our segmentation result will become the visited array we created to keep track whether a point has visited before. The BFS algorithm can be checked below;

```
 def BFS(self):
     """
     Traversing the points on the 3d grid by applying Breadth-first search.
     Initially, it loops over max number of seeds, then applying BFS starting
from
     that seed point.
     """
     for i in range(self.max_seed):
         start_node = self.findSeedPoints()
         if start_node == []:
             break
         self.pointMean = self.img[start_node[0], start_node[1]]
         self.checkedPoints = 0
         self.queue.push(start_node)
         self.visited[start_node[0], start_node[1]] = 1
         while not self.queue.isEmpty():
             temp = self.queue.front()
             self.queue.pop()

             for i in range(len(self.neighbour)):
                 new_node = [temp[0]+self.neighbour[i][0],
temp[1]+self.neighbour[i][1]]
                 if self.checkIfOut(new_node) and self.checkThreshold(new_node)
and not self.isVisited(new_node):

                     self.checkedPoints += 1
                     new_point = self.img[new_node[0], new_node[1]]
                     self.pointMean = (self.pointMean * (self.checkedPoints -
1) + new_point) / self.checkedPoints
```

```
                    self.queue.push(new_node)
                    self.visited[new_node[0], new_node[1]] = 1
```

4. Finally, we only calculate our dice score using our visited array, and ground truth provided before. The calculation of dice score;

```
def calculateDiceScore(self, ground_truth):
    """
    Calculate the Dice Score based in visited array and ground truth
    """
    a, _ = np.nonzero(ground_truth)
    b, _ = np.nonzero(self.visited)

    bool_gt = (ground_truth == 1)
    bool_seg = (self.visited == 1)

    intersection = (np.logical_and(bool_gt, bool_seg)) * 1.0
    c, _ = np.nonzero(intersection)
    return 2 * (len(c))/(len(a) + len(b))
```

For the 3D region growing, similar functions have been used with little modification, adding third axis to the calculations.

### Results

- For **Task 1**, we got an average dice score **0.85423** using threshold 0.125, 15 seeds at most, and 8-neighborhood.
- For **Task 2**, we got an average dice score **0.84277** using threshold 0.13, 15 seeds at most, and 4-neighborhood.
- For **Task 3**, we got dice score **0.89688** using threshold 0.3, 5 seeds at most, and 26-neighborhood.
- For **Task 4**, we got dice score **0.88056** using threshold 0.25, 5 seeds at most, and 6-neighborhood.

The result NIfTI files can be accessed from here